RL-TR-91-361
Final Technical Report
December 1991

VALIDATION OF KNOWLEDGE BASED SYSTEMS

Lockheed Missiles & Space Company, Inc.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. 6407

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-91-361 has been reviewed and is approved for publication.

APPROVED: Robert N. Ruberti

ROBERT N. RUBERTI
Project Engineer

FOR THE COMMANDER: Raymond P. Urtz

RAYMOND P. URTZ, Jr.
Director of Command, Control & Communications

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(C3CA) Griffiss AFB, NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

VALIDATION OF KNOWLEDGE BASED SYSTEMS

Contractor: Lockheed Missiles & Space Company, Inc.
Contract Number: F30602-88-C-0130
Effective Date of Contract: 30 September 1988
Contract Expiration Date: 30 December 1990
Short Title of Work: Validation of KBS
Period of Work Covered: Mar 89 - Jun 90

Principal Investigator: Dr. Rolf Stackowicz
Phone: (512) 448-5718

RL Project Engineer: Robert N. Ruberti
Phone: (315) 330-3528

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Robert N. Ruberti, RL (C3CA) Griffiss AFB NY 13441-5700 under Contract F30602-88-C-0130.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Final Mar 89 - Jun 90

4. TITLE AND SUBTITLE
VALIDATION OF KNOWLEDGE BASED SYSTEMS

5. FUNDING NUMBERS
C - F30602-88-C-0130
PE - 62301E
PR - F407
TA - 01
WU - 01

6. AUTHOR(S)
---

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Lockheed Missiles & Space Company
Software Technology Center
2100 East St Elmo Road
Austin TX 78670-7100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced
Research Projects Agency
1400 Wilson Blvd
Arlington VA 22209

Rome Laboratory (C3CA)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-91-361

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Robert N. Ruberti/C3CA/(315) 330-3528

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Knowledge-based systems (KBS) technology has emerged from applied Artificial Intelligence as a means for modeling complex systems. KBS's have consequently become more and more integrated into large complex software systems in defense, industry, business and science. Failures in these systems could be critical, but methods for validating conventional software systems do not take into account the incremental life cycles and other pecularities of KBS engineering. The objective of this contract was to define and develop a KBS validation system. DEVA, that enables a user to check the redundancy, consistency, completeness and correctness of DARPA-Sponsored KBS's. This report describes the implementation of the DEVA modules developed under this contract: a KEE translator, a rule refiner, a validation and verification of nonmonotonic reasoning module and structure, logic, extended structure, extended logic, semantics, omission, control checkers.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Rule-Based Systems, Validation, Verification, Structural Consistency Checking	142
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

Chapter 1

EXECUTIVE SUMMARY

The objective of this two and a quarter year contract (10/88-1/91) was to define and develop modules for validating knowledge-based systems (KBSs) that enable the user to check the redundancy, consistency, completeness and correctness of DARPA-sponsored KBSs, such as Air-Land Battle Management (ALBM), Pilot's Associate (PA), or Fleet Command Center Battle Management Program (FCCBMP), constructed with IntelliCorp's Knowledge Engineering Environment (KEE) expert system shell.

The system built, DEVA (DARPA Expert System Validation Associate), consists of ten modules, each with several submodules: the

KEE Translator,
Structure Checker,
Logic Checker,
Extended Structure Checker, (now the extended mode of the structure checker),
Extended Logic Checker, (now the extended mode of the logic checker),
Semantics Checker,
Omission Checker,
Rule Refiner,
Control Checker,
and a module for the Validation of Nonmonotonic Reasoning.

The nine validation modules are generic and can work with any translator.

The original implementation schedule was to deliver two of these modules during each 6-months contract phase; in addition to that the Extended Structure Checker and Extended Logic Checker were to be delivered at the end of Phase 2 (9/89). Problems in adequately staffing the effort resulted in a two months' delay, and Phase 2 was delivered 11/89. At the November 1989 review Mark Fausett, the technical monitor of the DEVA contract at Rome Laboratory, suggested requesting a no-cost extension to the contract. As of 7/20/90 Lockheed Missiles & Space Company, Inc. executed and submitted Modification P00003 (via LMSC Transmittal Letter, F404020) to extend the period of performance on the contract by three months to 12/30/90 to RADC, Directorate of Contracting.

While developing DEVA, we discovered that none of the DARPA projects mentioned above were still using KEE rules; some of them had completely abandoned the use of KEE. Our attempts to find KEE rule bases through the KEE Users Group were not successful. The DEVA modules were thus tested with in-house test systems and with large rule bases (about 1,000 rules) of varying depths generated with a synthetic rule-base generator built for this purpose. The avalability of these synthetic rule bases led to considerable performance improvements of the validation modules in comparison to the earlier in-house IR&D prototypes, in some cases up to 3 orders of magnitude. The genericness of the DEVA modules was successfully tested in combination with a translator — developed in MACProlog with internal Lockheed IR&D funds — for expert systems written in C Language Integrated Production System (CLIPS).

The Manned Maneuvering Unit Fault Diagnosis, Isolation and Reconfiguration (FDIR) expert system provided by NASA Johnson Space Center consists of 104 CLIPS rules. The

CLIPS translator produced 357 normalized internal rules which were checked by the structure and logic checkers. The structure checker found 4 redundant CLIPS rules, 9 unreachable predicates and many un-firable rules.

This report describes the scope of the effort and the design and implementation of the ten major DEVA modules listed above and their integration with the KEE environment.

Appendices A through C were provided with the first Interim Technical Report (June 1989). Appendix A contains the original DEVA User Guide describing the DEVA-KEE *interactive* mode where all user KEE transactions were automatically recorded, translated and sent to DEVA for loading and use while the user was working in KEE. This effective way of communication between the two systems had to be abandoned because the many different methods KEE uses for accessing and modifying the internal representation of a knowledge base made this mode unreliable. It was replaced in January, 1990 by the *batch* mode. (For details, see sections 4.2 and 4.3).

Appendix B discusses the applicability of the DEVA modules to the validation of both frame- and rule-based systems. Appendix C gives a brief history of approaches to the validation of knowledge-based systems.

We would like to express our appreciation to the many employees of IntelliCorp who willingly helped us to understand the intricacies of KEE in the process of translating KEE to DEVA. We would also like to thank the employees at Quintus Prolog for their support. We finally express our thanks to the employees of the Lockheed Artificial Intelligence Center and Software Technology Center who worked on this project.

CHAPTER 2

OVERVIEW: VALIDATION OF KNOWLEDGE-BASED SYSTEMS

2.1 PROBLEM

Knowledge-based systems (KBSs) technology has emerged from applied Artificial Intelligence (AI) as a vital technology for modeling complex systems. KBSs have consequently become more and more integrated into large complex software systems in defense, industry, business, and science. Failures in these systems could result in great danger to life and property; therefore, validation of KBSs is of utmost importance to software systems developers, implementors, and especially users.

The growing reliance on KBSs requires the development of appropriate methods for validating such systems, because the traditional validation methods used for conventional software are not directly applicable to KBSs. KBSs are typically developed and validated according to the so-called exploratory program development paradigm. In this development style, the expert or user states the requirements and the knowledge engineer represents the requirements as one or more rules, facts or objects, i.e., as executable specifications. These are compiled and executed by the inference engine of the development shell. The expert (knowledge engineer) then compares the actual output with the expected output. If the outputs disagree, the expert *debugs* the requirements or specifications.

This style of validation of KBSs is thus a process of incremental revision of the program logic which typically requires many iterations — with many false tries — through the development life cycle until the expert, user, or knowledge engineer is satisfied that the resulting program meets the requirements. This methodology, however, does not ensure the reliability of the created KBS, since the expert's requirements may still be imprecise, conflicting, partially wrong, or incomplete. In addition, the representation of the knowledge may similarly be imprecise, erroneous or inadequate.

An additional problem stems from the fact that KBS software — in contrast to traditional software which is procedural with explicit control flow within and between modules — is mostly declarative, with little or no explicit control and no explicit functional modules. KBS software lacks the explicit functional structure which is a prerequisite for tracing and validating requirements.

To validate and verify a KBS, we must be able to associate a knowledge base (KB) with the set of functional requirements it is supposed to implement. And similarly, we must be able to associate a particular requirement with the set of rules, facts, or objects which implement it.

In contrast to other current approaches of validating KBSs, which have not progressed during the last four years beyond the validation of basically syntactic properties of KBSs, this effort is based on an approach to validation that exploits more powerful information. It makes use of semantic information and meta-knowledge about the objects, actions, rules, control strategies, and integrated behavior of knowledge-based applications. It further exploits and makes explicit the structural relations immanent in KBSs.

2.2 GOAL AND OBJECTIVES

The stated objective of this contract was to define and develop a KBS validation system, DEVA (DARPA Expert System Validation Associate), that enables the user to check the redundancy, consistency, completeness and correctness of DARPA-sponsored KBSs (e.g., Air-Land Battle Management, Pilot's Associate, Fleet Command Center Battle Management Program) constructed with IntelliCorp's KEE expert system shell.

The functionality of DEVA, as shown in Figure 1, is contained in a wide range of generic validation tools which include the structure checker, logic checker, extended structure checker, extended logic checker, semantics checker, omission checker, rule refiner and control checker. The only DEVA module which is specific to KEE is the KEE translator which converts a KEE KBS into the internal data structures used by the DEVA validation tools.

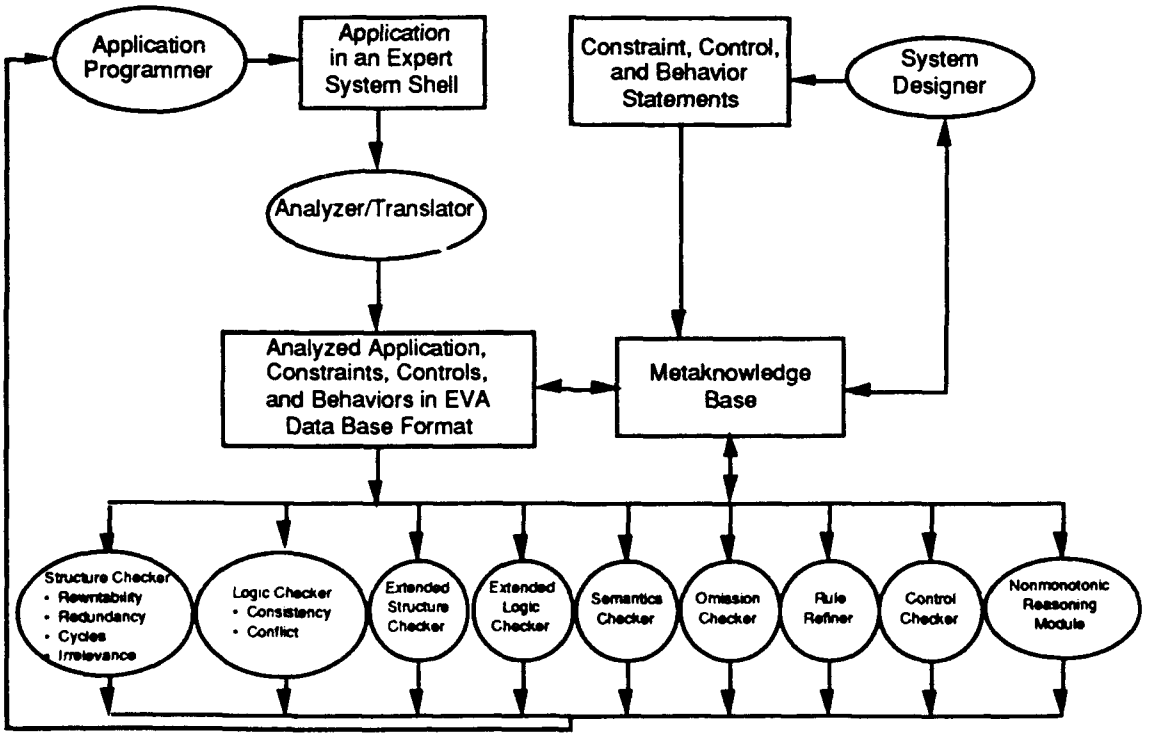

Figure 1

This Final Technical Report describes the implementation of the DEVA modules developed under this contract: the KEE Translator, Structure Checker, Logic Checker, Extended Structure Checker, Extended Logic Checker, Semantics Checker, Omission Checker, Rule Refiner, Control Checker, and the Validation of Nonmonotonic Reasoning module.

2.3 TECHNICAL APPROACH

The knowledge representation of rule-based and frame-based ES shells is logic-based. Applications or models written in such shells are therefore amenable to validation using logic and mechanical theorem proving. (Cf. Appendix B, The Validation of Frame-Based Systems.) The validation of such systems can be further improved by making use of meta-knowledge (higher-order knowledge) with semantic information and integrity constraints. This permits the system's designer to formulate conditions that the rules, frames, and facts of a domain model have to meet; it also permits the automatic checking of the validity of the underlying application without being limited to only those constraints that can be formulated in the language supported by the shell.

DEVA provides the system designer with just such a language to express the constraints which pertain to the domain model. DEVA then also uses the system designer's meta-statements for the validation of the KB. To safeguard against unwanted modifications of the knowledge base, DEVA **only makes observations and recommendations**. DEVA **warns, but does not modify** the KB.

Our approach for KBS validation is obtained through three important features of DEVA:

- a unifying architecture
- a common meta-language
- an integrated environment

2.3.1 DEVA Architecture

DEVA is built on a unifying, extensible platform. The unifying architecture is based upon

- A single user interface for all checkers,
- A single meta-knowledge base for all checkers,
- A common meta-language for specifying constraints.

All of the DEVA checkers are implemented in Quintus Prolog 2.5 and use the same DEVA internal data structures. The KEE-DEVA translator rewrites a KEE application KB into these data structures. Invoking a DEVA checker is thus comparable to entering a goal (query) in Quintus Prolog.

Our selection of Prolog as the implementation language was based on the following considerations:

- The validation of KBSs requires extensive automatic theorem proving. Prolog has a built-in automatic resolution-based theorem prover. By using Prolog we expedited the development of DEVA as well as provided more validation functionality within the proposed time frame since the automatic theorem prover did not have to be developed separately.
- Maintaining the meta-knowledge base requires the functionality of a database

management system. Prolog provides a built-in database management system, which made it unnecessary to develop such functionality separately.

- The meta-language required for representing validation criteria as meta-statements can be defined and implemented in Prolog. This made it unnecessary to develop a separate abstract machine to interpret the DEVA meta-language.

2.3.2 Meta-Language

Facts and rules in a KEE KB are object level (first-order) declarations. Validation thus means checking the correctness, consistency and completeness of a first-order KB. To permit validation using logic, automatic theorem proving, and semantics, validation criteria are defined in terms of the object level declarations, and are themselves meta-level (second or higher-order) declarations.

DEVA provides a declarative meta-language for the developer to specify the meta-level validation criteria. This meta-language permits the systems developer to define a wide range of conditions that must be met by the application KBS in order to be correct, consistent, and complete. These meta-level conditions are defined in much the same way as the developer defines the object level declarations: as special DEVA rules whose syntax is that of KEE rules. This will be discussed in more detail in Section 8 on the Semantics Checker.

The meta-language contains many *built-in* primitive meta-predicates which can be used to represent validation criteria. The developer only needs to know the meaning of these meta-predicates and how to use them.

The meta-predicates permit the developer to execute validation modules at different levels of detail. For example, the meta-predicate, *subsumes(< rule1> ,< rule2>)*, denotes that rule < rule1> is subsumed by rule < rule2> . Thus, we can enter

```
subsumes(Ri,Rj)
```

to find all rule pairs where the second rule subsumes the first, or enter

```
subsumes(Rj,rule_19)
```

to list all rules subsumed by *rule_19*, or test whether *rule_3* is subsumed by *rule_24* with

```
subsumes(rule_3,rule_24).
```

Parameters (arguments) to meta-predicates can be variables and constants, such as unit names, slot names, rule names, etc.

The long-range advantage of using a declarative language is that declarative languages are more amenable to execution on parallel architectures. Since Parallel Prolog systems are being developed elsewhere, our Prolog-based meta-language system can eventually make use of these Parallel Prolog systems for the validation of very large KBs.

2.3.3 System Integration

KEE and DEVA run as two independent processes. DEVA has an image of a KB in KEE. Whenever a KB in KEE is *sent* to DEVA, its image in DEVA is changed to reflect its current image in KEE. This way the KB in KEE and its image in DEVA are kept in sync.

In *Phase 2* we supported an interactive mode between KEE and DEVA such that whenever an operation such as *ADD*, *CHANGE.TO*, or *DELETE* was applied to a rule, a unit or a slot value, the operation was performed by KEE, and then a message containing the translated information was sent to DEVA to update its image. However, we learned that we could not always predict the behavior of KEE and were thus forced to abandon this interactive approach. (For details, see section 4.2 of this report.)

KEE and DEVA each have their own full-screen desktop window. At any time, there is only one forefront desktop which accepts characters from the keyboard. The background desktop can be brought to the front by clicking a left mouse button on its desktop icon. The user will usually invoke DEVA from KEE and DEVA will return any error/warning messages after it executes the validation modules on the image of the loaded KB. The display of the error/warning messages for the majority of the checkers is text-oriented at this time, but in future versions these messages may also be graphic-oriented (e.g., by highlighting the incorrect/questionable rules). The interface for the *Phase 3* module, the Rule Refiner, which is described in detail in section 10.3 of this report, is an example of a graphic-oriented interface.

2.4 ADVANTAGES OF DEVA APPROACH

We regard the following as the main advantages of the DEVA approach.

- the DEVA approach is general and not limited to a particular shell, like all the other KBS validation approaches described in the literature. (Cf Appendix C. A Brief History of Validation of Knowledge-Based Systems.)

- DEVA is not limited to the validation of propositional logic like many other approaches.

- DEVA uses a logic-based declarative metalanguage which permits the statement of requirements and constraints in a logical formalism and their verification and validation by means of mechanical theorem proving.

- The DEVA language is more expressive than other shell languages and thus permits the formulation of many requirements and constraints that cannot be made in those languages.

- The smooth integration of development environment and DEVA environment make DEVA a natural extension of the development environment.

CHAPTER 3

DEVA USER INTERFACE

3.1 DEVA AND KEE FROM THE SAME DESKTOP

This section assumes some knowledge of the Sun Windowing System known as Sunview or Suntools. If unfamiliar with Sunview or Suntools, read the Sunview Manual before proceeding. From the UNIX prompt type:

```
sunview
```

or

```
suntools
```

To load DEVA, open a shell window and enter:

```
deva.exe
```

To load KEE, from another shell window type:

```
kee
```

When the KEE desktop appears, it will take several minutes to load the ADVICE system (used by the DEVA translator). During the KEE loading process, the user will be prompted in the lower left corner typescript window:

```
Do you wish to load the Translator (y/n)?
```

The user should enter *Y*, and hit return.

3.2 THE DEVA WINDOW

The DEVA User Interface uses the unified design principle. This principle states that there is generally a short list of guidelines that can be learned through interaction with the interface, and once learned be applied through out the interface. The standard DEVA browser is one example of this principle.

In SunView, the functionality of an interface object is sometimes ambiguous. Take the example of a browser, which can look exactly like a picture with only one scroll bar. To distinguish between them, when a browser is used in the DEVA interface it is peach colored, a simple but effective visual clue.

The DEVA window can be divided into four sections:

Menu bar
: A set of horizontal labels in which each label activates a unique pop-up menu (also known as a pull-down menu). The pop-up menu is activated by moving the cursor to

a menu label and holding down the right most mouse button (also known as a *right click and hold*). While holding down the right mouse button, the user moves the cursor vertically over the pop-up window. As the cursor passes over a menu option, it is inverted. To select a menu item, the user lets up on the right mouse button while that menu item is inverted.

Report Window
: Output of all checkers are sent to this window.

Message Window
: Messages generated by the system are sent to this window.

Options Window
: Frequently user-modified options are contained in this window.

3.2.1 Menu Bar

The Menu bar contain the following entries:

- Structure
  - Dead-End
  - Unreachability
  - Subsumption
  - Implication
    - Redundancy via BC
    - Redundancy via BC-CYC
    - Redundancy via QSQR
    - Redundancy via Residues
  - Cycle Detection
  - Irrelevance
  - Indirect Irrelevance
  - Ambiguity
- Logic
  - Rule Inconsistency
    - Inconsistency via BC
    - Inconsistency via BC-CYC
    - Inconsistency via Residues
  - Add/Delete Ambiguity
    - Add/Delete via BC
    - Add/Delete via BC-CYC
    - Add/Delete via Residues
  - Conflict
    - Conflict via BC
    - Conflict via BC-CYC
    - Conflict via QSQR
  - Add/Delete Conflict
    - Add/Delete Conflict via BC
    - Add/Delete Conflict via BC-CYC
    - Add/Delete Conflict via QSQR
- Semantic
  - Inverse
  - Interstate Integrity

Min-Max-Set
Subrelation
Value

Completeness
Frame Omission
MetaKB-Based Class Suggestions
Rule-Based Class Suggestions
Incomplete Relation Taxonomy
Incomplete Slot Values
Rule Omission
Logical Completeness
Numerical Completeness
Suggest Rules by Analogy
Refine Rules

Control
Control Constraints
Sequence
Exclusion
Necessity
Conditional
BC Interference

Nonmonotonic
Useless NWA Rules
Impossible Plans

Utilities
Connection Graph
Display Frame
Statistics
Save Logfile
Unit Graph
Residue Static Relations

3.2.1.1 Functionality of the Utilities pull-down menu

Connection Graph
Builds a Connection Graph for the current KB (consult 2.3.1 Connection Graph).
Display Frame
allows the user to display Frame knowledge in the report window.
Statistics
display statistics about the current KB (number of rules, lhs clauses,...) in the report window.
Save Logfile
writes the contents of the report window to ./deva.log.
Unit Graph
Builds a Unit Graph for the current KB (consult 2.3.2 Unit Graph).
Residue Static Relations
Put up a browser to Define Static Relations.

3.2.2 Options Window

The Options Window contains the following items

Text items:

A text item is used for input from the user. The text item is displayed as a label (identifying what the text item stands for) and sometime contains a user-editable value.

Version:
shows the version numbers of DEVA, Quintus-Prolog, and Quintus-Prowindows. It is not user-editable.

Knowledge Base: < KB Name>
contains the name of the KB loaded and is not a user-editable value. A right click, hold on < KB Name> , and selection of *Rename Knowledge Base* allows the user to change the name of the current KB.

File: < File Name>
Name of file to store the KB in DEVA format. If < File Name> is not specified when the user left clicks on the *Save* button, a dialog box will be opened prompting input.

Infer Depth: < integer>
Depth at which to terminate search. Default value is 2.

Buttons:

A button is displayed as a rounded-corner box with a label inside. The button is activated by a left click on the box's label. The button responds by flashing the rounded-corner box, and calling the function programmed to handle it. Sometimes a button will have a pop-up menu associated with it that can be activated by a right click hold on the button's label. If no pop-up menu exist, a right click will do nothing.

Load
The *Load* button has a pop-up menu associated with it which contains the following items:
Load KB in DEVA Format
Load KB in KEE-messages Format

Save
Save a KB to disk in DEVA format.

Help
Place DEVA in help mode.

Quit
Exit DEVA.

Erase
Erase the current KB from DEVA's active memory.

Clear
Clear the Report Window.

Browse
Lets the user display a rule, constraint rule, KEE ValueClass, or Proof Tree from a DEVA browser. If the user highlights a rule name, or constraint name before left clicking the *Browse* button, the text of the rule or constraint pops up in its own window. If no selection is highlighted, or the selection is not a rule or constraint,

then the rule browser pops up. The *Browse* button has a pop-up menu associated with it which contains the following items:

Browse Rule

To display a KEE Rule, pull down the pop-up menu associated with the *Browse* button and select *Browse Rule*. This will cause a DEVA browser to appear. By choosing one of the labels in the browser (left clicking to select), the selected KEE Rule will be displayed in its own pop-up window.

Browse Constraint

To display a DEVA Constraint Rule, pull down the pop-up menu associated with the *Browse* button and select *Browse Constraint*. This will cause a DEVA browser to appear. By choosing one of the labels in the browser (left clicking to select), the selected DEVA Constraint Rule will be displayed in its own pop-up window.

Browse KEE ValueClasses

To display KEE ValueClass information, pull down the pop-up menu associated with the *Browse* button and select *Browse KEE ValueClasses*. This will cause a DEVA browser to appear. By choosing one of the labels in the browser (left clicking to select), the selected KEE ValueClass information will be displayed.

Browse Proof Trees

Within the textual message of anomaly warnings will be references to proof-tree labels. Valid labels (not just including labels that are relevant to redundancy checking) include redundancyX, rederivableX, residue-redundancyX, conflictX, add-delete-conflictX, rule-inconsistencyX, residue-inconsistencyX, add-delete-ambiguityX, etc. To graphically browse a mentioned proof tree, pull down the pop-up menu associated with the *Browse* button and select *Browse Proof Trees*. This will cause two windows to appear. The window entitled *DEVA Proof Selector* is a browser menu listing all proof-tree labels which have appeared in the messages diplayed in the report window. By choosing one of these labels (left clicking to select), the selected proof-tree can be browsed in the window entitled *DEVA Proof Browser*.

The *DEVA Proof Browser* window is divided into 3 regions. The upper left right hand corner region lists the current goal stack (stack grows downward). Initially the top of the goal stack will be a term indicating the purpose of the proof tree (i.e. redundant(r1), rederivable(r1), rule-inconsistency, conflict, etc). An uninstantiated forward-chaining copy of the rule used to prove the goal at the top of the goal stack is displayed in the upper left hand corner region. Initially this will be a Horn Logic encoding of the anomaly definition proved. For example, the following top-level rule would be used to express that a redundancy had been found in rule r1:

```
rule name : deva default constraint
   absent(rule(r1)),
   provable(A)
   B
   = >
   redundant(rule(r1))
```

The lower region contains an instantiated backward-chaining copy of the rule displayed in the upper left hand region. The lower region rule also contains a hypertext capability to allow further browsing of the subgoals. By holding down the right mouse button on a subgoal, one will either see *assumed true* indicating if the proof strategy assumed it true; *true via fact lookup* indicating a fact was used to prove the goal; or *show proof-tree* to indicate that a rule was used to prove this subgoal and that further browsing into the proof-tree is possible from this point. If one selects the *show proof-tree* message when it appears, all of the following occur: the selected subgoal is added to the top of the goal stack (upper right hand region) ; an uninstantiated copy of the rule used to prove the new subgoal is displayed in the upper left-hand region; and an instantiated version of the rule now appears in the lower region. Further browsing can now be done with respect to this new proof context.

When one is finished browsing at some context in the proof tree and does not wish to (or is unable to) delve deeper into the proof tree, simply select one of the goals on the ancestor stack (upper right hand region) to reset the DEVA Proof Browser to that ancestoral context. To browse another proof tree entirely, simply select another label from the *DEVA Proof Selector* window. Hit the *Cancel* button on the *DEVA Proof Selector* window to terminate all proof-browsing.

Interrupt
Stop currently running checker.

Circular Buttons:

A circular button is a menu in which the current choice is displayed. When the user left clicks on a circular button, it cycles to display the next choice. A right click hold on a circular button, displays a pop-up menu of all the possible choices, the current choice being checkmarked.

Data
KEE Messages
Allow user to load KBs saved in DEVA messages format.
DEVA Format
Allow user to load KBs saved in DEVA format.

Mode
Regular
Do not use metarelationships between units.
Extended
Use metarelationships between units.

3.3 DEVA GRAPHICS

3.3.1 Connection Graph

When the user pulls down the DEVA Utilities menu and selects *Connection Graph*, a graphical representation of the KB's Connection Graph is built in a newly created window called *DEVA Connection Graph*. Rules are represented by a blue outline box with the rule-name contained

inside, and are referred to as a *rule node*. Dependencies between rules are represented by red arcs connecting the rule nodes. For example, an arc pointing from Rule1 to Rule2 would mean there is a literal in the right hand side (RHS) of Rule1 that matches (unifies with) a literal in the left hand side (LHS) of Rule2.

3.3.1.1 Connection Graph Layout Algorithm

The Connection Graph is layed out in levels (top to bottom) according to the number of rules a particular rule *triggers* (also called RHS to LHS connections). Rules at the top of the graph trigger more rules than those at the bottom. Each level is then layed out (left to right) according to the number of rules it is *triggered by* (also called LHS to RHS connections). The algorithm is the Woods graph layout algorithm for hierarchical graph algorithm for directed graphs that was adapted by Rowe et. al. This algorithm has several advantages:

The complete rule name is contained within the rule node box.

Rules that have no connections to other rules are placed on the bottom level of the graph.

On each level adjacent rule nodes have their Y coordinates offset so that overlapping arcs are minimized.

3.3.1.2 Connection Graph Browser

The left side of the Connection Graph contains a browser (known as the *Connection Graph Browser*) which contains a sorted list of all rules contained in the KB. A left click on a rule name in this browser displays the text of the rule in a pop-up window, just as a left click on the rule node in the Connection Graph would. A middle click arranges the graph so that the clicked on rule node is against the left most edge of the graph. This functionality is useful for locating a rule node in a very large graph.

Above the Connection Graph Browser are two buttons, *Quit* and *Close*. *Quit* destroys the Connection Graph, and *Close* iconifies it.

Right click hold on a rule node produces a pop-up menu which allows highlighting of certain rule relations. Each relation (triggers or triggered by) has a sub-menu of unique colors that are used in highlighting a relation in the Connection Graph. Only one relation per node is allowed. The following options are available on this rule relation menu:

triggered by
: highlights all nodes that point to this node.

triggers
: highlights all nodes that this node points to.

remove highlighting
: unhighlights the displayed.

Other mouse button options are:

Middle click
: allows the rule node to be dragged to another location.

Left click
displays the text of the rule clicked on in a pop-up window.

3.3.2 Rule browser

The rule browser is a list of the current rule names in the system, and it is located to the right of the Connection Graph. Left click on a rule name displays the text of the rule clicked, just like a left click on a rule node. Middle click moves the object to the very left of the visual graph. With a very large graph this helps the user locate a particular rule node.

3.3.3 Unit Graph

When the user pulls down the DEVA Utilities menu and selects *Unit Graph*, a graphical model of the fact base domain is built in a newly created window called *DEVA Unit Graph*. A Unit Graph is a dynamic undirected acyclic graph that presents the relational hierarchy of class, subclass, and instance links in the knowledge base. One may examine the slots and slot values for all units in the Unit Graph to determine their relevancy to the rules.

In the Unit Graph, class-units are light blue; member-units are either light-grey (unhighlighted) or yellow (highlighted); the background is light grey; the lines that represent the relation between a class-unit and its subclass-unit are solid scarlet, and the line that represents the relationship between a class-unit and its member-units is a dashed line of the same color.

A left click on a member-unit toggles it's status (yellow = highlighted, light-grey = unhighlighted). A left click on a class-unit toggles all of its members (including member-units of its subclasses).

A middle click and drag on any unit allows the user to reposition the unit within the graph. A right click hold on a class-unit displays the slots that are defined at that class in a pop-up menu in which the menu-items are slot names.

When sub-menus (also known as walking menus) appear in the pop-up menu, the item(s) that appear in a sub-menu are that slot's default values. When the class-unit has one or more subclasses, the names of the subclasses are contained in the sub-menu of the menu-item *subclass*. This feature is quite useful since it shows the KEE ordering in which the subclasses were defined. Similar to a class-unit right click hold, a member-unit right click displays that member-unit's slots for which it has a value. The sub-menu's menu-items are that slot's value(s).

3.4 SOFTWARE AND HARDWARE REQUIREMENTS

Software:
Sun Microsystems Common Lisp version 2.1.3
IntelliCorp KEE version 3.1.75
SunOS 4.03
deva.exe (the runtime version of DEVA)

Hardware:

- Sun 4 or 3
- Color Monitor
- 3 button Sun mouse and pad

CHAPTER 4

THE KEE TRANSLATOR

4.1 INTRODUCTION

In Phase 3 the KEE to DEVA Translator was modified to run in batch mode only. The interactive mode of the Translator was dropped from the DEVA system in January, 1990. Prior to that time, as the user worked in the KEE environment, all KEE transactions were automatically recorded, translated and sent to DEVA for loading and use. The interactive mode proved to be unreliable because of the many different methods KEE uses for accessing and modifying the internal representation of a knowledge base. This is discussed more fully in sections 4.2 and 4.3.

In the current version of DEVA the user has the option of loading the DEVA Translator when the KEE environment is entered. Once the user has loaded the DEVA Translator, he has the option to unload or reload the Translator at any time during the session via the KEE Icon Command Menu. While the Translator is active (loaded) the user may translate a KEE KB into KEE message format by selecting the *Send KB to DEVA* item from the KB Command Menu. The resulting translation is sent to a DEVA process via Unix pipes (see section 4.4). This batch translation also takes place automatically whenever the user saves the current version of an application KB.

The Translator also automatically maintains a special constraint KB for every application KB that a developer loads or creates in the KEE environment. Just as the developer uses the application KB to model the problem domain, he uses the corresponding constraint KB to model the specifications pertaining to the problem domain. These specifications are defined as DEVA metarules in a KEE-like syntax to facilitate their construction by the developer. (These metarules are described in detail in Section 8 on Semantic Constraints). When a KB is loaded, the Translator automatically also looks for a *-constraint* KB to load (e.g., if the KB file is named *auto-repair* then the corresponding constraint file is named *auto-repair-constraint*). If one does not exist then a new constraint KB is created and loaded automatically.

The KEE KB Command Menu system has been modified so that the user may perform only limited (appropriate) actions on constraint KBs. For example, the user may only create rule classes and display the constraint KB. He may not save, delete, create, copy, or rename the constraint KB. All of the above mentioned actions happen automatically to the constraint KB when the user applies them to the associated application KB. For example, whenever a KEE KB source file, in a *-constraint.u* file, and a DEVA translation of both the application KB and its corresponding constraint KB is saved in a *-kee.msg* file.

4.2 THE ADVICE SYSTEM

The interface between the Translator and KEE is implemented using the Sun Lisp ADVICE system. With ADVICE one may wrap user-defined code around any Lisp function body. When ADVICE is attached to a function name, the user-defined code for that function is executed rather than the original function body. Within the user-defined ADVICE code one may call the original function body by using the function *apply-advice-continue*.

In DEVA 2.0, when we were supporting an interactive mode between the Translator and KEE, we attached ADVICE to all of the known KEE KB management functions (e.g.,

create.slot, *delete.slot*, *delete.rule*, *create.unit*, *rename.unit*, etc.) in order to capture any KB activity. However, we continued to experience different and unexpected KEE KB management behavior depending on the order of KB editing events.

KEE is a very large system with incomplete documentation. It was impossible to guarantee that all operations in KEE which have an effect on data and rules were being covered in the interactive mode; therefore, in Phase 3 we elected to abandon the interactive mode of the Translator, and focus on the batch mode.

In batch mode Sun Lisp's ADVICE function continues to be attached to the major KEE KB management functions: *load.kb*, *create.kb*, *delete.kb*, *rename.kb*, and *save.kb*. These ADVICED functions allow us to maintain a corresponding semantic constraint KB for every application KB in the KEE system (see section 4.6).

4.3 KEE

IntelliCorp does not maintain a formal grammar for the KEE language (KEE users are provided with an interface for creating KEE KBs). Thus the Translator's grammar was created through observation, and may not be complete. The translation is actually performed using KEE functions and predicates. That is, the Translator does not process ".u" files (KEE KB source files), but rather it uses KEE access functions such as *kee::units.kb* and *kee::unit.slots* etc., to retrieve information about KEE KBs. This information is then translated into Prolog predicate output. The Translator creates a string of all of the translated functions and predicates and writes them to a file.

Again, it must be stressed that KEE has no formal syntax or semantics, and the informal syntax and semantics (in the form of user tutorials and tutorial-like documentation) provide examples of what can be done in KEE, not a complete description of the KEE language.

4.4 THE COMMUNICATIONS SYSTEM FROM THE KEE PERSPECTIVE

Communication between KEE and DEVA is now possible through Unix pipes. This new functionality replaced the file based message system in previos versions of DEVA. The pipe connection not only makes for faster communication between KEE and DEVA, but allows DEVA to send information back to KEE, and even query KEE for specific information. Additionally, it is possible run KEE and DEVA on different systems and still communicate and the Unix pipe communication makes it easier to integrate DEVA with other systems such as ABE.

4.5 THE KEE TRANSLATOR

The Translator uses various KEE access functions to read in data from the KBs that have been loaded into KEE. Currently, the KEE Translator reads in and produces output for the following KEE objects: KBs, units, rules, slots and facets. The Translator reads in this information and produces Prolog as its output. It produces a predicate called *message*. The first argument of *message* is the action that DEVA is to take. In batch mode this argument is always *create*. The second argument of the message is the translation, when appropriate, of the KEE object to which the action is to be applied, and the third argument is a time stamp. For example, the following message:

```
message(create, rule('division',
    [[text,['start','constraining','the','mission','of',Vx,[]]],
     [slot, 'phase.info', Vx, Vy],
     [slot, 'unit.being.supported', Vy, Vz],
     [slot, 'unit.being.supported', Vz, Vm],
     [slot,'mission',Vy,Vym],
     [slot,'mission',Vz,Vzm],
     [slot,'mission',Vm,Vmm],
     [kee_equal,[equal,Vvar2,[lisp,['set-difference',Vym,Vzm,Vmm,[]],[]],[]],[]],[]],
    [[change_to, [slot,'mission',Vy,Vvar2],[]],[]],

     'same.world.action','div-rules','friendly_units',

    ['(','IF',
     '(','START','CONSTRAINING','THE','MISSION','OF','?X',')',
     '(','?X','IS','IN','CLASS','FIGHTERS',')',
     '(','THE','PHASE.INFO','OF','?X','IS','?Y',')',
     '(','THE','UNIT.BEING.SUPPORTED','OF','?Y','IS','?Z',')',
     '(','THE','UNIT.BEING.SUPPORTED','OF','?Z','IS','?M',')',
     '(','THE','MISSION','OF','?Y','IS','?YM',')',
     '(','THE','MISSION','OF','?Z','IS','?ZM',')',
     '(','THE','MISSION','OF','?M','IS','?MM',')',
     'THEN',
     '(','CHANGE.TO',
        '(','THE','MISSION','OF','?Y','IS',
             '(','SET-DIFFERENCE','?YM','?ZM','?MM',')',')',')',')']),

     [17,56,11,24,7,1990]).
```

tells DEVA to create a rule called *division* with the left hand side

```
(START CONSTRAINING THE MISSION OF ?X)
(X IS IN CLASS FIGHTERS)
(THE PHASE.INFO OF ?X IS ?Y)
(THE UNIT.BEING.SUPPORTED OF ?Y IS ?Z)
(THE UNIT.BEING.SUPPORTED OF ?Z IS ?M)
(THE MISSION OF ?Y IS ?YM)
(THE MISSION OF ?Z IS ?ZM)
(THE MISSION OF ?M IS ?MM)
```

and the right hand side

```
(CHANGE.TO
   (THE MISSION OF ?Y IS
      (SET-DIFFERENCE ?YM ?ZM ?MM)))
```

The main access functions are:

kb.name - to get the name of a kb;
kb.units - to get all the units in a kb;

unit.name - to get a unit's name;
unit.slots - to get the slots in a unit;
unit.parents - to get a unit's parents;
unit.children - to get a unit's children;

slot.name - to get a slot's name;
slot.values - to get a slot's values;
slot.min.cardinality - to get the minimum cardinality of a slot;
slot.max.cardinality - to get the maximum cardinality of a slot;

get.value - with argument *rule.type* - to get a rule's type;
get.value - with argument *premise* - to get a rule's premise;
get.value - with argument *conclusion* - to get a rule's conclusion;
get.value - with argument *external.form* - to get a rule's text representation;

wff.to.list - to convert a KEE well-formed formula (WFF) structure into a Lisp list.

4.6 DEVA SEMANTIC CONSTRAINTS

DEVA also manipulates the constraint KBs that a developer uses to create DEVA metarules (These metarules are described in detail in Section 8 on Semantic Constraints). When a KB is loaded, the Translator automatically looks for a "-constraint" KB to also load (e.g., If the KB file is named *auto-repair* then the corresponding constraint file is named *auto-repair-constraint*). If one does not exist then a new constraint KB is created and loaded automatically. A special KEE menu for constraint KBs is provided so that the user may perform only limited (appropriate) actions on a constraint KB. For example, the user may only create rule classes and display the constraint KB. He may not save, delete, create, copy, or rename the constraint KB. All of the above mentioned actions happen automatically to the constraint KB when the user applies them to the application KB with which the constraint KB is associated. In other words, the constraint KBs are automatically maintained by the DEVA Translator.

4.7 RESTRICTIONS AND EXCEPTIONS

DEVA 2.0 did not properly handle variable names (that portion of a variable ID following the mandatory question mark) which contained a character other than an alphanumeric character or underscore (i.e., ?big-ship was not allowed) nor an apostrophe in a text string (i.e., "let's see if he's home" was not allowed). The problem was not with the Translator, but with the manner in which Prolog interprets delimiters in variables and double and single quotes. These two restrictions were eliminated in Phase 3. The variable problem was rectified such that nonalphanumeric/nonunderscore characters in a variable are changed to underscore while at the same time preserving the uniqueness of the variable within the scope of the rule. The apostrophe problem was handled by scanning the KB input and removing all apostrophe's within a text string prior to translation.

While eliminating these two previous restrictions, we have had to introduce another: using a quoted variable with KEE's *equal* operator. There is a bug in the WFF representation of KEE's *equal* operator. When the second argument to the *equal* operator is a quoted atom whose first character is a question mark (KEE's variable format), the quote is not present in the WFF. Although the WFF representation is the same whether the second argument is quoted or not, the behavior of the rule literal the WFF represents is not. During rule

execution the quoted atom is not evaluated and the unquoted variable is. IntelliCorp has been advised of this problem.

4.8 THE NORMALIZER

KEE permits rules with logical *or*'s like

R1: if or([a(X),b(X)]), c(X) then d.
R2: if or([g(X),a(X)]), c(X) then d, e.

Given the facts *a(i)* and *c(i)*, where *i* is an arbitrary constant, both rules *R1* and *R2* would apply.

The function of the normalizer is to convert the KEE rules — which KEE converts internally to Conjunctive Normal Form (CNF) — to Disjunctive Normal Form (DNF); this facilitates the recognition of rule duplication and subsumption.

The KEE rules above would thus be converted to the following DEVA rules:

R1.1: if a(X), c(X) then d.
R1.2: if b(X), c(X) then d.
R2.1: if g(X), c(X) then d, e.
R2.2: if a(X), c(X) then d, e.

where the subsumption of *R2.2* by *R1.1* is now easily recognizable.

This section briefly describes some of the simplifications that KEE performs when creating CNF's, the additional DEVA reductions, and the creation of the DNF's.

When a user enters a rule in KEE it is stored in the user's exact form in the slot *External Form*. KEE performs the following simplification on the LHS of the rule and converts it into CNF before it stores the result in the *Premise* slot.

(not (cant.find A)) is converted to *(find.any A)*
(not (find.any A)) is converted to *(cant.find A)*
(not (not A)) is converted to *A*
(cant.find (cant.find A)) is converted to *(find.any A)*
(find.any (cant.find A)) is converted to *(cant.find A)*
(cant.find (find.any A)) is converted to *(cant.find A)*
(find.any (find.any A)) is converted to *(find.any A)*
(< literal> or < literal>) is converted to *(or < literal> ,< literal>)*
(< literal> and < literal>) is converted to *(< literal> ,< literal>)*
(and < literal> < literal>) is converted to *(< literal> ,< literal>)*

DEVA performs additional reductions using De Morgan's formulas and equivalences among the KEE operators *find.any*, *cant.find*, and *for.always*.

(find.any (A or B)) is converted to *(or (find.any A) (find.any B))*
(cant.find (A or B)) is converted to *(and (cant.find A)(cant.find B))*
(for (A or B) always C) is converted to *(and (for A always C) (for B always C))*
(for (A always B)) is converted to *(cant.find (and A (cant.find B)))*.

When all literals have been reduced, the rule is in atomic form. DEVA constructs the DNF representation of the rule from the conjunction of disjunctions which is KEE's CNF representation of the premise.

For example, suppose the KEE user types in the rule:

```
((IF (?F IS IN FLOWER)
   (OR (AND (THE COLOR OF ?F IS RED)
            (THE STEM_TYPE OF ?F IS THORNY))
       (AND (THE COLOR OF ?F IS YELLOW)
            (THE STEM_TYPE OF ?F IS THORNY))
       (AND (THE COLOR OF ?F IS PINK)
            (THE STEM_TYPE OF ?F IS THORNY)))
 THEN (?F IS IN ROSE)))
```

KEE stores the premise of the rule in a nonminimal CNF:

```
((member_of(F, flower),
 (or(color(F, pink),
    color(F, yellow),
    color(F, red))),
 (or(color(F, pink),
    color(F, yellow),
    stem_type(F, thorny))),
 (or(color(F, pink),
    stem_type(F, thorny),
    color(F, red))),
 (or(color(F, pink),
    stem_type(F, thorny),
    stem_type(F, thorny))),
 (or(stem_type(F, thorny),
    color(F, yellow),
    color(F, red))),
 (or(stem_type(F, thorny),
    color(F, yellow),
    stem_type(F, thorny))),
 (or(stem_type(F, thorny),
    stem_type(F, thorny),
    color(F, red))),
 (or(stem_type(F, thorny),
    stem_type(F, thorny),
    stem_type(F, thorny)))),
```

DEVA converts this representation to a DNF representation using the algorithm of Slagle, Chang and Lee which removes nearly all of the redundancies introduced by KEE's normalization to CNF ("A new algorithm for generating prime implicants", *IEEE Transaction on Computers*, Vol. C-19, No. 4, pp.304-310, April 1970). The final step of the DEVA normalizer prunes the few remaining redundant LHSs before storing the rules. The result is

three DEVA rules with the left-hand sides:

```
[member_of(Vf,flower), color(Vf,red), stem_type(Vf,thorny)],
[member_of(Vf,flower), color(Vf,yellow), stem_type(Vf,thorny)],
[member_of(Vf,flower), color(Vf,pink), stem_type(Vf,thorny)]]
```

CHAPTER 5

STRUCTURE CHECKER

5.1 DEVA DATA STRUCTURES

All DEVA validation modules make use of the internal DEVA data structures created from the translations of the KEE units and rule-units.

A KEE rule is internally represented by means of four DEVA structures:

rule(Rule_name, LHS, RHS) rule_type(Rule_name, Rule_type) rule_class(Rule_name, Rule_class) rule_kb(Rule_name, Rule_kb)

where *LHS* is a *wff* in a conjunctive normal form [Chang and Lee 1973], *RHS* is a list of actions, *rule_type* is one of *new_world_action*, *same_world_action*, or *deduction*, and *rule_class* and *rule_kb* are user-assigned names. The rule_type, rule_class, and rule_kb information is extracted from the corresponding KEE rule slots.

The following KEE rule:

```
(RULE1
  (IF (IN.CLASS ?CAR CARS)
      (OWN.VALUE BATTERY ?CAR DEAD)
   THEN
      (ADD (IN.CLASS ?CAR NON.FUNCTIONING.VEHICLES))))
```

is thus translated into the DEVA structure:

```
rule(rule1,
    [cars(Vcar),battery(Vcar,dead)],
    [non_functioning_vehicles(Vcar)])
```

5.2 CONNECTION GRAPHS

Connection graphs are very important data structures which are used by DEVA to analyze KBSs. A connection graph is defined as follows: an arc in the connection graph denotes a match between a literal in the LHS of a rule and a literal in the RHS of a rule. A fact is considered a rule without a LHS.

In order to build efficient meta-interpreters, DEVA maintains data-structures that can quickly retrieve information about the connection graph for the checker modules. These data-structures have been devised to make full use of the Warren Abstract Machine's indexing on the predicate name and the first argument. For example:

rule(?RuleId,?LHS,?RHS)

is used to retrieve the LHS and RHS of a specific rule. The RHS data-structure:

rhs(?Literal,?Rule_Id)

is used to quickly index to a rule containing ?Literal on its RHS. A RHS literal such as *add(not(a()))* would actually be stored in the database as :

rhs2(a(X),add(not(_)),ruleId1)

thereby allowing indexing on predicate name *a* of arity 1. This was done because the external wrappers, such as *add(not(_))*, provide very little indexing benefit. The LHS data structure:

lhs(?Literal,?RuleId)

is used to quickly index to a rule containing ?Literal on its LHS. The following data structure:

rhsTOlhs(?Rule1,?Rule2)

indicates that there exists a RHS literal in ?Rule1 which satisfies a LHS literal in ?Rule2.

Using the connection graph, it is trivial to implement a depth-first backward chaining meta-interpreter (but it is not guaranteed to terminate). For example:

```
bc(Goal):-
  bc2([Goal]).

bc2([]).
bc2([Goal|Rest]):-
  fact(Goal),
  bc2(Rest).
bc2([Goal|Rest]):-
  rhs(Goal,Rule_Id),      % find a rule to prove Goal
  rule(Rule_Id,LHS,RHS),  % Gather LHS SubGoals
  member(Goal,RHS),       % Propagate Variable Bindings
  bc2(LHS),
  bc2(Rest).
```

5.3 DEADEND RULES

A rule is called a *deadend rule* if its LHS can never be satisfied. This happens when the LHS of the rule contains a literal which cannot match with any fact or any literal in the RHS of a rule.

For checking deadend rules, we introduce the meta-predicate *deadend* defined as follows:

deadend(< rule_id> , < LHS_literal>).

This predicate denotes that rule < rule_id> is a deadend rule because of the unsatisfiability of the LHS literal < LHS_literal> . In terms of the connection graph: a rule is a deadend rule if it contains a LHS literal that is not pointed to by any arc.

5.4 UNREACHABLE FACTS AND LITERALS

A fact or a RHS literal of a rule is called an *unreachable literal* if it cannot be matched by any

LHS literal of a rule. Thus unreachable literals can not be used and may be omitted or the KBS may require rule modification.

To find unreachable literals, we use the meta-predicate *unreachable* defined as:

unreachable(< rule_id> , < RHS_literal>).

This predicate denotes that the RHS literal < RHS-literal> of rule < rule_id> is unreachable. In terms of the connection graph: a fact or RHS literal of a rule is unreachable if it is not pointed to by any arc.

5.5 REDUNDANCY

A large KB may be developed by more than one person. When different subsets of the KB developed by different persons are merged, there may be overlaps and redundancies. The redundancies may cause performance, maintenance, updating and synchronization problems. For example, given two rules *R1* and *R2*, if the LHS of *R1* subsumes the LHS of *R2*, and if forward chaining is used, then whenever *R2* can be fired, *R1* also can be fired. Therefore, any actions (*ADD*, *DELETE* or *CHANGE.TO*) in the RHS of *R2* which also occur in the RHS of *R1* will be redundant. This causes a performance problem because the same actions are repeated. The same redundancy may also cause an updating problem because desired behaviors may not be achieved if *R1* and *R2* are not updated at the same time. The synchronization problem may occur if different actions in the RHSs of both *R1* and *R2* need to be taken simultaneously. In this last case, *R1* and *R2* should be combined into one rule to prevent them from firing at different times.

We consider three types of redundancies: *Duplication*, *Subsumption*, and *Implication*. The latter is also referred to as *Indirect Subsumption*. Duplication and subsumption redundancies involve only two rules, while implication redundancy involves more than two rules.

Two methods for detecting Implication Redundancy are discussed. The first approach, the restricted generate-and-test method, generates an incomplete, but revealing, set of fact-base scenarios to check whether redundancies stem from those scenarios. The second approach, the residue-analysis method, is more general and attacks the problem from the reverse angle. Rather than generating situations and checking for problems, we also ask "What must be true of the fact-base in order for redundancy to exist? Furthermore is this fact-base likely to occur?".

5.5.1 Direct Subsumption and Duplication

This section describes direct subsumption involving two rules. Within this section, subsumption means direct subsumption. Subsumption occurs between two rules when one rule fires every time the other rule fires and produces all the same conclusions (maybe more).

In this case the first rule totally subsumes the function of the second rule and the second rule can be omitted. Duplication is a special case of subsumption where one rule fires every time the other rule fires (but NOT more often) and produces all the same conclusions (and NOT more). Since duplication is "subsumed" by subsumption, duplicate rules will automatically be detected by the subsumption checker. Here is an example of subsumption:

R1: A $\rightarrow$ X,Y,Z

R2: A,B → X,Y

Rule *R1* will fire every time *R2* fires since its LHS is a subset of *R2's* LHS and will deduce everything *R2* deduces, therefore, Rule *R1* subsumes Rule *R2*.

Considerable progress has been made under this contract in performance improvements for subsumption checking. Using specially constructed data structures, and several effective heuristics, the check was sped up by a factor of 200 over the brute force method. DEVA can now check a KB of 500 rules for subsumption in under 4 seconds (on a Sun 4/110c running Quintus Prolog 2.5).

In general, the most useful of these optimizations is the addition of a simple data structure that allows extremely fast indexing of literals into the rules in which they occur. The data structures, one for LHS literals and one for RHS literals, are basically hash-tables from literals to rule ID's. In the brute force method, to find which rule(s) contains a literal, one needs to look through each rule searching the LHS or RHS for the literal. With the new indexing structures, the Prolog system hashes to the rule(s) involved almost instantly. Since it is exactly this type of lookup that most of the checkers rely on, a dramatic and comprehensive performance improvement was realized. In the case of subsumption checking, we used the heuristic that a subsumed rule shares at least one RHS literal with any possible subsuming rules, so given any rule, finding the set of rules with which it shares a RHS literal involves a quick lookup through the hash-table for RHS literals. All but a relatively small set of candidate rules are immediately screened from the search.

Once a pair of candidate subsuming/subsumed rules has been gathered, the LHSs and RHSs must be checked for the proper subset relationship. If *R1* subsumes *R2*, *R1's* LHS literals will be a subset of *R2's* LHS literals (requiring fewer conditions to fire), and *R1's* RHS literals will be a superset of *R2's* RHS literals (producing more results when it does fire). To optimize this process, a special subset procedure was written that aborts the check as soon as an element is found which fails the membership test.

At this level in the subsumption checking process, literals from different rules are being compared to each other to see if they match. This matching process is called unification, but unification is not a sufficient test by itself to guarantee subsumption. For example in the rules

R1': a(X,Y) → p(X,Y).
R2': a(X,Y) → p(Y,X).

R1' does not subsume *R2'*, even though rule *R1'* unifies with *R2'*. Since *R2'* also unifies with *R1'*, one might be tempted to think these two rules are duplicates. But consider the effect of these two rules on the following data:

F1: a(red,blue).

Rule *R1'* produces p(red,blue), but *R2'* produces p(blue,red).

We have identified several other situations where special attention must be paid to the variable bindings when checking for subsumption:

a) Constants in the potentially subsuming rule would prevent the two rules from producing

all the same results. Consider

R1': a(X,5) → p(X,5), q(X,5), r(X,5)
R2': a(X,Y), b(X,Z) → p(X,Y), q(X,Z)

Unlike the structurally identical cases, *R1'* does NOT subsume *R2'* because, although the LHS of *R1* unifies as a subset of *R2's* LHS, it will not actually fire in all the same instances. Furthermore, *R1* produces only facts whose second argument is *5* while *R2* can produce any value in that position.

b) Natural unification of variables results in differing conclusions. Consider

R3: a(X,Y) → p(X,Y)
R4: a(X,X) → p(X,X)

Here the LHS of *R1'* unifies with the LHS of *R2'*, but as variables bind and propagate across the implication sign, the result does, in fact, turn out to be different.

c) Information gathered in the extra literals on the LHS of the "subsumed" rule is needed in the conclusions on the RHS. Consider

R1': a(X,Y) → p(X,Y), q(X,Y), r(X,Y)
R2': a(X,Y), b(Y,Z) → p(X,Y), q(Y,Z)

R1' can't duplicate the effect of *R2'* since whatever data would bind to *Z* in the literal b(Y,Z) is not available to *R1'*

5.5.2 Implication Redundancy Checking via a Restricted Approach

Before proceeding, we would like to point out that all the results of this section, 5.5.2, are necessary background material for rule-inconsistency detection in Chapter 6 Logic Checker.

A rule *R1*: (LHS = > RHS) is redundant if the action(s) on the rule's RHS can be independently derived from the knowledge base **and** the condition(s) in the LHS of *R1*, **but without using rule** *R1* in the inference process. In other words, it may be possible to exclude the rule from the knowledge base or, at the very least, replace the redundant rule with a version that does not include the redundant RHS action. For example, consider the rules :

R1: A → B
R2: B → C
R3: B,X → C
R4: A → C

Rule *R4* is redundant. To see this:

a) Assume *R4*'s LHS is provable. Imagine literal A is a fact.

b) Exclude rule *R4* from the knowledge base.

c) Check to see if rule *R4*'s RHS action, C, is derivable.

Notice that rules *R1* and *R2* interact to derive C. Rule *R4* is unnecessary. Rules *R1* and *R3* do not make rule *R4* redundant, since there exists an extraneous condition X that needs satisfaction.

In general this is an intractable problem. The brute force approach to solving this problem would be to generate the power set *P* of all rules in the knowledge base. Each member *p* of *P* would then be considered to see if the rules in *p* could interact to cause a redundancy in some other rule *r* in the KB, where *r* is not an element in *p*. The size of the power set is exponential in the number of rules. Obviously this is undesirable. Due to the general intractability of the problem there does not exist any one good solution. A very straightforward/elegant strategy may perform quite admirably on some *reasonable* search-spaces, but will very quickly become overwhelmed when dealing with a more *hostile* search space. Also a more sophisticated strategy, which does a better job on hostile KBs, can have a huge overhead which makes it less appropriate for dealing with a KB with a *friendly* search space. The approach taken by DEVA is to provide a suite of algorithms from which the user can choose. These will be discussed in more detail later in this section.

On the surface it seems that redundancy is merely an efficiency problem, since results may be re-derived. It is, however, also potentially dangerous, especially when dealing with a knowledge base containing retractions and assertions on the RHS of a rule. There may be a synchronization problem. For example, consider:

R11: rank(?x,private), deserves_promotion(?x) →
delete(rank(?x,private)),add(rank(?x,corporal))

R12: rank(?X,private), deserves_promotion(?x) →
delete(rank(?x,private))

If rule *R12* is fired before rule *R11*, then there is the possibility that *R11* will not be satisfied. This is because rank(?x, private) is no longer true when rule *R11* is checked for satisfiability, since it has been deleted.

Redundancy also makes the task of maintaining a KB more difficult. For example: the person in charge of KB maintenance modifies the KB (adds, deletes, or modifies a rule) in order to achieve a different behavior in the expert system, but the modification does not alter the behavior. The problem could be that the modification involved a redundant rule; therefore, the addition and/or deletion of the rule had no effect on the performance of the expert system. Tracking these problems down can be very time consuming, and thus add significantly to the lifetime cost of an expert system.

Skolemization is required to correctly check a first-order logic knowledge base for redundancy. Variables and constants within the arguments of a first-order literal may make the literal less general than another similar literal. For example, the literal *loves(?X,?Y)* is more general than *loves(?X,?X)*. All solutions for the subgoal *loves(?X,?Y)* will contain all the solutions for *loves(?X,?X)*. So in the case of redundancy detection, if there is a way to prove independently *add(loves(?X,?X))*, then it is not necessarily true that *add(loves(?X,?Y))* is also provable. The standard way to determine if one literal subsumes another is to use skolemization. Skolemization is a process that replaces the variables contained in the literal with constants. For example, the literal L2: *loves(?X,?Y)* is skolemized to *loves(sk1,sk2)*. The literal L1: *loves(?X,?X)* is skolemized to *loves(sk1,sk1)*. If a skolemized literal L1 can unify with another (unskolemized) literal L2, then L2 subsumes L1.

Thus a new definition is needed for Redundancy in the context of first-order logic knowledge bases.

Definition:

K = The set of static facts and rules in the knowledge base
R = The rule being checked for redundancy.
K' = K set minus R
R' = R after undergoing skolemization
V = assumed set of valid input facts (LHS of R')
A = RHS of R'.

Redundancy exists if $V \cup K'$ entails a, such that a is a member of A.

Example:

R2: a(?X), b(?X) → c(?X)
R3: a(3) → c(3)

R2': a(sk1), b(sk1) → c(sk1)

Rule *R2* above is not redundant. It is not guaranteed that rule *R3* will fire whenever rule *R2* does, rather only when ?X is bound to *3*. Rule *R2'* represents the skolemized version of *R2*, where all variables have been turned into skolem constants. It is now obvious that there is no way to independently cause the assertion of *c(sk1)* via rule *R3* and assumed facts *a(sk1)* and *b(sk2)*.

5.5.2.1 What does restricted redundancy detection involve?

Redundancy checking requires inference. Either forward or backward chaining can be used to accomplish the task. Neither solution is absolutely better than the other. Depending upon the characteristics of the KB's search space, one approach may be better than the other. Results, outside this realm of validation, by Mike Genesereth from Stanford University have shown that inference using **both** backward chaining and forward-chaining can be superior to committing exclusively to either one or the other.

Three top-down backward-chaining algorithms have been implemented for redundancy. The performance of these algorithms can be greatly enhanced by the storage of a provability cache, whose size is not dependent upon the size of the extensional fact-base. The differences between the algorithms will be presented later, as well as a detailed discussion of the provability cache in section 5.5.2.3.

An outline of an elegant solution using backward-chaining was presented by the DEVÁ team member C.-L. Chang. Essentially the idea involves translating forward-chaining rules to backward-chaining Prolog rules. Consider this sample knowledge base:

R111: a(?X),b(?Y) → add(c(?X,?Y))
R112: a(?X1) → add(d(?X1))
R113: b(?Y1) → add(e(?Y1))
R114: d(?X2),e(?X2) → add(c(?X2,?X2))
R115: e(?X3) → add(not(c(?X3,?X3)))

and the resultant Prolog program:

R111p: c(X,Y):- a(X),b(Y).
R112p: d(X1):- a(X1).
R113p: e(Y1):- b(Y1).
R114p: c(X2,X2):- d(X2), e(X2).

% example of explicit negation
R115p: not(c(X3,X3)):- e(X3).

Since the rule-set [*R112p,R113p,R114p*] unioned with the assumed input fact-set [*a(sk1),b(sk2)*] entails proof of the literal *c(sk1,sk2)*, then rule *R111* is redundant. An algorithm which models this approach via meta-interpretation has been included in the suite of redundancy algorithms. The details of this algorithm are given in section 5.5.2.2.

An extension of the above approach is required to correctly handle negation by failure during redundancy checking. When the checker reports that a rule is redundant, it must be redundant under all possible states of the KB. Therefore, a special treatment of negation by failure is necessary. In the following example, *R21* does not subsume *R31* (make *R31* redundant). However if *cant_find* was handled the same as negation by failure, then *R31* would be mistakenly labeled as redundant. This is because the rule-set [*R21p*] unioned with the assumed input fact-set [*a,d*] entails proof of literal *c*.

Example:

R21: A,cant_find(B) → add(C)
R31: A,D → add(C)

Resultant Prolog Program:

R21p: c:-a, unprovable(b).
R31p: c:-a,d.

During redundancy checking (as well as rule-inconsistency checking in the Logic Checker), negation by failure is handled by an explicit proof that the literal has been deleted. In other words, the checker will act as if the following Prolog rule were always present:

negByFail: cantfind(X):- delete(X).

Consider the following example of redundancy involving retractions on the RHS of the rule:

R41: A,cantfind(B) → delete(C)
R42: A,D → delete(C)
R43: A,cantfind(B) → add(D).

Resultant Prolog Program:

R41p: delete(c):- a,cantfind(b).
R42p: delete(c):- a,d.
R43p: d:- a,cantfind(b).

Since the rule-set [*negByFail,R42p,R43p*] unioned with the assumed input fact-set [*a,cantfind(b)*] entails proof of the literal *delete(c)*, then rule *R41* is redundant.

A further extension to the backward-chaining approach is necessary to completely and correctly handle numerical comparisons. It is possible for numerical comparisons to be included in the LHS of a rule which is being checked for redundancy. Assumed satisfiability of a numerical comparison containing a variable can entail proof of many other numerical comparisons containing the same variable. For example, *greater(?X,10)* entails *equal(?X,29)*, *notequal(?X,19)*, etc.

Definition:

Assume C1 and C2 are numerical comparisons which each contain one variable.

Range-subsumes(C1,C2) is true if the range of C1's variable is a superset of the range of C2's variable.

Theorem:

If numerical comparison C2 range_subsumes numerical comparison C1, then C2 is satisfiable whenever C1 is.

The range of *?X* in *greater(?X,10)* is *(10 .. PosInfinity]*, while the range of *?X* in *lessthanorequal(5,?X)* is *[5..PosInfinity)*. Since *[5..PosInfinity]* contains *(10..PosInfinity]*, we know *lessthanorequal(5,?X)* is true whenever *greater(?X,10)* is true.

A module has been introduced into DEVA to handle these types of proofs. Both the Redundancy and Inconsistency modules make use of this functionality. In DEVA, all numerical comparisons are enclosed within a *lisp(arity1)* term. Therefore the redundancy checker must act as if the following Prolog rule were present:

```
rangeSub: lisp(NumComp):-
              lisp(AssumedNumComp),
              range_subsumes(NumComp,AssumedNumComp).
```

For example, rule *R52* is made redundant by rule *R51*. This is because the rule-set [*rangeSub,R51p*] unioned with the assumed input fact-set [*age(sk1,sk2),lisp([greater,sk2,10])*] entails proof of the literal *quality1(sk1)*.

R51: age(?Person,?X),lisp([lessthanorequal,5,?X]) → quality1(?Person).
R52: age(?Person,?X),lisp([greater,?X,10]) → quality1(?Person).

Resultant Prolog Program:

R51p: quality1(Person):- age(Person,X), lisp([lessthanorequal,5,X]).

5.5.2.2 Restricted redundancy detection algorithms

In the previous section, the requirements of redundancy detection were described in the context of a backward-chaining algorithm involving a straight translation down to Prolog, but inference in Prolog is undecidable and incomplete. Therefore, we have implemented the model in the form of various meta-interpreters thereby allowing more control. A maximum inference depth can be specified for all of the algorithms thus allowing the user to specify the

degree of completeness he/she desires, while remaining decidable. Furthermore, the depth bound is intuitively desirable. We argue that it is more useful to first detect and remove anomalies involving a shallow inference chain before moving on to more complex and deep inference chains, since the longer inference chain may be a compounded side-effect of one or more smaller ones.

The next enhancement is to maintain the goal stack. As soon as a recursion is detected, the path is pruned. This has the benefit of guaranteeing termination, but it also makes the detection process incomplete. Redundancies introduced via recursion/cycles will not be detected. On mechanically generated knowledge bases with *hostile* search-spaces, significant performance gains were realized with a minimal amount of incompleteness.

Backward-chainers suffer in that they will cheerfully re-attempt or re-prove a task they have already completed. If one were to introduce dynamic programming, already proved/failed/in-progress goals could be memoized. Potentially this could save considerable work.

Work already exists in the database community to efficiently handle recursive queries in the presence of large amounts of data. Good solutions for recursive query processing can be modified to also work for redundancy/inconsistency detection in the presence of a hostile KB. The goals are similar: namely to ensure termination, to work with large amounts of data, and to be efficient.

The Recursive Query SubQuery algorithm is essentially a depth-first set-at-a-time (as opposed to Prolog's tuple-at-a-time) backward-chainer that does not backtrack and also uses dynamic programming. It is guaranteed to terminate on all queries in the domain of predicate logic where arguments are restricted to be co stants or variables (also known as DataLog). Potentially unsafe queries, such as *greater(X, 0)* which has infinite solutions, are handled in a special manner.

DEVA uses a limited form of this algorithm that inherits the termination guarantee. However, it will not find all redundancies and inconsistencies introduced via recursion. Instead only those resulting from one application of the recursion will be found. In addition, the algorithm has been modified to be also constrained by a maximum-depth bound. Essentially the modified algorithm remembers all goals that it has been asked to prove. It also remembers all intermediate derived relations. If an already seen goal should arise again, the memoized results are re-used. If there are no memoized results for the already-seen goal, then it assumes the original goal must have failed.

The overhead of the query subquery method is quite high. On especially hostile search spaces, it may be the best solution when working in conjunction with the provability cache.

5 5.2.3 The usefulness of a provability cache

The idea behind the provability cache is to provide a means of knowing beforehand which paths in the search-space are useless and therefore can be pruned away at runtime. Significant performance gains will be realized if such a mechanism can be integrated into the top-down strategies presented in the previous section. The worst case storage requirement of the provability cache is $O(N^2)$, where N is the number of unique rule identifier's in the knowledge base. This represents the ultimate in hostility among search-spaces.

In redundancy checking, inference is done over each assumption set. These assumption sets are simply the LHSs of rules. The cache provides pruning information for each assumption set. The provability cache is actually a collection of sub-caches. Each rule in the KB is assigned a sub-cache, with the sub-cache's assumption set being the rule's LHS. Each individual sub-cache, C, contains the rule identifier's of other rules which are likely to be satisfiable whenever the assumption set is satisfiable (with the restriction that the parent rule be excluded from all inference). Inclusion in a sub-cache does not guarantee satisfiability of the included rule, but all rules which are derivable will be included in the sub-cache.

In addition to stating what is likely to be proven and under what circumstances, the individual sub-caches also maintain an underestimate of how much effort is required. For example if *R3* is contained in rule *R1*'s sub-cache with an underestimated depth of 4, then that signifies that a proof tree (exclusive of rule *R1*) of at least depth 4 will be required to prove rule *R3* satisfiable via the assumption that rule *R1*'s LHS is satisfiable.

Standard Artificial Intelligence textbooks point out the advantage of using underestimates during search. If we know we are willing only to expend effort *E* to accomplish some task and the underestimate is greater than *E*, then we can prune away that branch in the search space. Therefore we are achieving two levels of pruning. First, no rule will be considered in a proof unless it is in the relevant provability sub-cache. Second, no attempt will be made to prove a rule if its underestimate is greater that the amount of allocated effort.

Computation of the provability cache is accomplished via the following:

```
For each rule, R, in the KB Do
  Begin
   - Assume that all LHS preconditions of rule R are provable via a fact lookup.
   - set 'NewRules' = [R]
   - set 'Depth' = 1
   Repeat
     - Find all Rules currently not in this provability cache that contain a
       LHS precondition that is satisfied
       by a RHS action in 'NewRules'. These are the 'CandidateRules'.
     - Set 'NewRules' = []
     - Set 'World' to be the set of static base and assumed facts UNIONED
       with the set of RHS literals of rules currently in the cache provided
       the included rule's inference depth is less than 'Depth'. This extra
       constraint on the depth is done to guarantee correct computation of the
       underestimated depth.
     - For each member C of 'CandidateRules'
            If C's LHS is directly satisfiable via a fact
              lookup from the 'World' set
            Then  add C to the R's sub-cache with an
                  underestimated inference depth set to
                  'Depth'.
                  Also add the satisfied rule to 'NewRules'.
     - Set Depth = Depth + 1.
   Until No new rules are added to R's Cache
  End {For Do-Loop}
```

We can now revisit the simple meta-interpreter by enhancing it to use a provability cache and

a maximum depth specifier.

```
% prove a goal with respect to rule ?ID's Valid Input Set.
% Limit the inference Depth to ?MaxD
bc(Goal,Id,MaxD):-
  bc2([Goal],Id,0,MaxD).

bc2([],_,_,_).
bc2([Goal|Rest],Id,CurrentD,MaxD):-
  fact(Goal),   % This will succeed on the LHS assumption set also.
  bc2(Rest,Id,CurrentD,MaxD).
bc2([Goal|Rest],ID,CurrentD,MaxD):-
  rhs(Goal,RuleId),
  provable(ID,RuleId,Effort), %Attempt to Prune
  Remaining is Effort +  CurrentD,
  Remaining = <  MaxD,
  rule(RuleId,LHS,RHS),  % Gather LHS SubGoals
  member(Goal,RHS),      % Propagate Variable Bindings
  NewD is CurrentD +  1,
  bc2(LHS,Id,NewD,MaxD),
  bc2(Rest,Id,CurrentD,MaxD).
```

5.5.3 Implication Redundancy Checking via Residue Analysis

The residue-analysis technique was originally explored to handle the needs of the Control Checkers (Chapter 11). Since we determined that it could also be used to provide a truly general form of redundancy and rule-inconsistency checking we implemented an implication redundancy checker via Residue Analysis. We briefly discuss this method and show that redundancy and rule-inconsistency was enhanced as a byproduct of the work on the Control Checker. (The paper "Uncovering Redundancy and Rule-Inconsistency in Knowledge Bases via Deduction" presented at the Fifth Annual Computer Assurance Conference, *IEEE COMPASS-90*, discusses these implementations in more detail.)

The truly general way to check for anomalies is to consider all possible fact-base scenarios, and then check for anomalies. In general this is not computationally practical, and semantically it may not make sense to consider a particular fact-base scenario. Just because some combination of facts is a member of the universe of all fact-base scenarios does not necessarily mean it will ever occur in practice. There are many laws of the domain, unknown to the knowledge base, which restrict combinations of facts.

In the restricted generate-and-test approach, a finite set of likely fact-base scenarios is checked for the occurrence of anomalies. Because the fact-base scenarios are determined by the preconditions of rules, all derived anomalies should be taken seriously. Afterall, the developer who wrote the rules determined that the preconditions were likely to occur together. Due to the incompleteness of fact-base scenarios considered, many anomalies can go undetected.

We have also implemented another more general approach, using proof residues, which can detect anomalies in fact-base scenarios not considered by the restricted generate-and-test approach. Rather than detect anomalies from specified fact-base scenarios, a residue approach will try to derive a fact-base scenario which supports a specific anomaly. "What must be true

of the fact-base for the specified anomaly to exist, and are these conditions likely to occur simultaneously?"

The proof-residue of some goal is the set of conditions missing from the fact-base which are needed before one can prove the goal. To determine all the proof residues for a goal, one needs to generate all possible proof-trees of the goal (down to a specified depth, to ensure termination). The leaves of the proof-tree, which must be assumed true (i.e., the residue), constitute a fact-base scenario, and when added to the KB entails the goal. The intermediate nodes and the root of the proof-tree are all ramifications of the assumed fact-base scenario. It is important that the proof residues and their ramifications be consistent.

To show redundancy in the rule R1: LHS → RHS, **independently** (not using R1) derive a residue proof for a RHS consequent, then check if the nodes in the reside proof tree entail all literals in LHS. If the residue proof tree entails LHS, then the rule R1 will be satisfiable whenever the residue proof tree is.

In those cases where an entailment relationship does not exist, but for which other possible proofs exist to independently derive the goal, a browsing capability can be used to explore the residues.

5.6 CYCLES

Due to the declarative nature of rules comprising a knowledge base, unforeseen patterns of interaction between the rules may arise during run-time. One such pattern is a cycle. In a forward chaining system, each of the actions on the RHS of a rule may cause a condition on the LHS of another rule to be satisfied. Each one of these potential *rhsTOlhs* connections is captured in the connection graph model of the rule base. A cycle exists when a RHS action in Rule *R1* directly or indirectly causes satisfaction of a LHS condition in Rule *R1*. In one case, such as a model of a feedback system, this may be exactly what is desired. In another case, the interaction pattern may be unwanted and dangerous (i.e., preventing termination). In either case, the detection of the cycle will prove useful to the knowledge engineer. In the first case, the message will re-affirm the knowledge engineer's intentions. In the second case, a dangerous interaction will have been unearthed.

Cycle-detection is an intuitively easy concept to grasp, but a difficult goal to accomplish efficiently while remaining complete. The obvious solution to generating potential cycles is as follows:

> For every Rule *ri* in the rule-base Do
> enumerate all paths from *Ri* to *Ri* using the cached *rhsTOlhs* connections in the connection graph

This extremely inefficient algorithm also suffers from the defect that it will report the same potential cycle *N* times, where *N* is the length of the cycle in question. The candidate cycles detected will not visit an intermediate rule more than once. For example, the cycle *a* to *b* to *c* to *b* to *a* would not be generated, since it visits rule *b* twice. Instead we would detect *a* to *b* to *a* and *b* to *c* to *b*.

5.6.1 Efficient Non-Redundant Cycle-Detection

Cycle detection is a two step process. The first step is the on-the-fly generation of a safe and

efficient connection graph. Safety is achieved by finding a set of edges all of whose removal would make the graph acyclic. These cycle-producing edges will be referred to as ancestor edges. For each detected ancestor edge, at least one unique cycle exists that uses that edge. The efficiency is gained by labeling the edges in a way that prevents redundant detections of the same cycle. The product of this first-step is a depth-first spanning forest representation of the connection graph. Once this has been created, the cycles can be extracted in a very efficient manner.

5.6.1.1 Creation of the depth-first spanning forest

The algorithm to create the depth-first spanning forest is an extended version of a simpler algorithm found in Aho, Hopcroft, and Ullman's *The Design and Analysis of Computer Algorithms*.

Input: A graph G = (V,E) represented by adjacency lists L[v].
- V is the set of rules.
- E is a set of *rhsTOlhs* connections.
  L[v] will be the set of rules S such that an *rhsTOlhs* connection exists from v to s, where v element of V and s element of S.

Output: A set of Trees that comprise the spanning forest.
The edges in each tree will be partitioned into:
- T, a set of tree edges. All vertices contained in the set T are considered "owned" by the Tree.
- Seen, a set of edges that lead to vertices which had been previously encountered along some other path. Note that the edge might lead to a node owned by a different tree.
- Ancestors , a set of edges such that if all where removed the connection graph would be acyclic.

Complexity: Order O(E) Time complexity, where E is the number of edges

```
PROCEDURE df_spanning_forest:
 begin
    FOR all v in V Do mark v "unseen";
    WHILE there exists a vertex v in V marked "unseen" DO
     begin
        Create new tree Ti with v as the root;
        Path =  path created by adding v to the empty path;
        df_edge_labeling(v,Path,Ti);
     end
  end
```

```
PROCEDURE df_edge_labeling(v,CurrentPath,Tree):
  begin
     mark v "seen"
     Indicate that v is "owned" by Tree
     FOR each vertex w on L[v] DO
        IF w is marked "unseen" THEN
           begin
              add (v,w) to T, the set of tree-edges in Tree ;
              NewPath =  path created by adding w to CurrentPath;
              df_edge_labeling(w,NewPath,Tree);
           end
        ELSE
           begin
             IF w exists in CurrentPath THEN
                add (v,w) to Tree's Ancestors set;
             add (v,w) to Tree's Seen set;
           end
  end
```

5.6.1.2 Using the spanning forest to enumerate the cycles

Once the forest is created, cycle-checking proceeds quickly and smoothly. Each tree in the spanning forest is searched in a depth-first manner to enumerate the cycles. When performing the depth-first search of each tree in the forest, the current path is maintained for the purpose of extracting the cycles.

An interesting property of cycles allows a good deal of pruning.

Theorem:
Vertices within a cycle (where each intermediate vertex appears once) will be "owned" by the same spanning-tree. This is true by the behavior of depth-first behavior of the procedure df_edge_labeling.

Imagine we are searching tree *Ti* and we encounter the edge (v,w). If *w* is "owned" by another tree *Tj*, then we know that we can prune away the subtree rooted at vertex *w*. This is because the edge leads to a vertex in a different spanning tree.

```
PROCEDURE all_cycles:
    FOR each tree Ti in the Spanning Forest DO
     begin
        v =  root(Ti);
        Path =  path created by adding v to the empty path;
        cycle_detect(v,Path,Tree);
     end
```

```
/* Searches all edges */
PROCEDURE cycle_detect(v,CurrentPath,Tree):
  begin
     FOR each vertex w on L[v] DO
        IF (v,w) is an Ancestor edge in Tree THEN
          Try to extract a cycle from CurrentPath;
        ELSE IF (v,w) is a Tree edge in Tree THEN
              begin
                NewPath =  path created by adding w to CurrentPath;
                cycle_detect(w,NewPath,Tree);
               end
             ELSE IF w is "owned" by Tree THEN
                  begin
                    NewPath =  path created by adding w to CurrentPath;
                    cycle_detect2(w,NewPath,Tree);
                   end
   end

/* Only search Seen and Ancestor edges */
PROCEDURE cycle_detect2(v,CurrentPath,Tree):
  begin
     FOR each vertex w on L[v] DO
        IF (v,w) is an Ancestor edge in Tree THEN
          Try to extract a cycle from CurrentPath;
        ELSE  IF (v,w) is a Seen edge in Tree) AND
                w is "owned" by Tree THEN
                   begin
                     NewPath =  path created by adding w to CurrentPath;
                     cycle_detect2(w,NewPath,Tree);
                   end
  end
```

When expanding out of a node that was reached via a tree Edge, all edges from the node will be searched. When expanding out of a node reached via a Seen edge, only the already-seen edges and ancestor edges from the node will be searched. It is this activity which prevents the generation of duplicate cycles.

Note that it is unnecessary to check for cycles at each step in the depth-first traversal. We know which edges are ancestor edges. Furthermore, we know that each cycle in the graph uses one of our ancestor edges. Thus checking for cycles need only occur after we have traversed an ancestor edge.

5.6.2 Possible Future Enhancements

This algorithm discovers minimal cycles which do not visit an intermediate node twice. The algorithm will be enhanced to report patterns of interactions between minimal cycles. For instance if we have detected minimal cycles (a-b-a) and (c-a-c), we can report that the two simple cycles have the capability of interacting and thereby creating a larger cycle. This can be done by detecting an intersection condition between the discovered minimal cycles.

A KB could potentially contain many cycles. To avoid overwhelming the user with cycle

warnings the above algorithm permits the user to group cycles according to whether they use the same ancestor edge or whether they involve a specific rule.

An addition to the Structure Checker's cycle detection was developed under the Control Checker task, and is discussed in section 11.6 Enhanced Cycle Checker.

5.6.3 Efficiency Gains

The algorithm discussed in the previous section is a significant improvement over the original version in the cycle-checker. The overhead of translating the connection graph into a non-persistent (i.e., not cached) depth-first spanning forest is definitely worth the effort, even though the process takes up most of the time required to run the cycle-checker. On small test KBs (i.e., 50 rules), the new algorithm is at least as fast as the original version. On larger test KBs, the new algorithm is the clear winner. It is difficult and useless to try to quantify how much better the new algorithm is based on one sample run, since the results are highly dependent on the characteristics of the KB (size, branching factor). To get a rough idea, on a KB of 400 rules the new algorithm took 8 seconds versus 32 seconds for the original version cycle-checker. On a KB with different characteristics, the improvement could be more dramatic.

Close examination of the above algorithms reveals them to be side-effect and data-structure driven. These algorithms do not fit neatly into the standard Prolog pattern-matching paradigm. One simple solution would be to perform all the side-effects by asserting and retracting to the database. In general this is an expensive and theoretically unclean technique. Studies by Richard O'Keefe from Quintus Computer Systems proved that side-effect driven algorithms can usually be implemented more efficiently without resorting to using the database as a scratchpad. Rather than use the underlying Prolog database, the depth-first spanning forest was implemented as a 5+ 5 tree. (An N+ K tree is a tree constructed of nodes that contain *N* data elements and *K* child pointer.) This permits access logarithmic (to the base 5) of the size of KB when retrieving the adjacency list of some rule.

5.7 IRRELEVANT CONDITIONS

A rule establishes a relationship between conditions in the LHS and conclusions in the RHS of the rule. That is, the rule specifies the dependency of the conclusions on the conditions. However, since the rule may not be exact, it may contain unnecessary or irrelevant conditions. A condition may specify a slot value. One way to tell if a slot value has any effect on a conclusion is to test if the same conclusion can be derived regardless what the slot value is. For example, consider the following two rules:

```
(RULE1
 (IF (IN.CLASS ?CAR CARS)
     (NOT (OWN.VALUE ENGINE ?CAR CRANKING))
     (OWN.VALUE GAS.TANK ?CAR EMPTY)
  THEN
     (OWN.VALUE BATTERY ?CAR DEAD)))
```

```
(RULE2
  (IF (IN.CLASS ?CAR CARS)
    (NOT (OWN.VALUE ENGINE ?CAR CRANKING))
    (NOT (OWN.VALUE GAS.TANK ?CAR EMPTY))
  THEN
    (OWN.VALUE BATTERY ?CAR DEAD)))
```

Clearly, the condition on whether the gas tank is empty or not has nothing to do with the conclusion that the battery is dead.

To check the existence of irrelevant conditions, we use the following algorithm:

a) Consider all rules in a KB one by one. Let *R* be a rule chosen from the KB.

b) If *R* does not contain negative literal, go to Step (a) to choose another rule. Otherwise, focus on a negative literal *L* in *R*.

c) Obtain R' from *R* by deleting the negation sign from *L*. If R' is directly subsumed by a rule *T* in the KB, then rules *R* and *T* contain the irrelevant conditions *L* and (NOT *L*).

We use the above example to illustrate the algorithm. RULE2 contains two negative literals. Let us first consider (NOT (OWN.VALUE ENGINE ?CAR CRANKING)). We obtain RULE2' from RULE2 by deleting the negation sign:

```
(RULE2'
  (IF (IN.CLASS ?CAR CARS)
    (OWN.VALUE ENGINE ?CAR CRANKING)
    (NOT (OWN.VALUE GAS.TANK ?CAR EMPTY))
  THEN
    (OWN.VALUE BATTERY ?CAR DEAD)))
```

Since RULE2' is not directly subsumed by RULE1, we do not find an irrelevant condition yet. However, if we consider the negative literal (NOT (OWN.VALUE GAS.TANK ?CAR EMPTY)), then we obtain the following RULE2" from RULE2 by deleting the negation sign from (NOT (OWN.VALUE GAS.TANK ?CAR EMPTY)):

```
(RULE2"
  (IF (IN.CLASS ?CAR CARS)
    (NOT (OWN.VALUE ENGINE ?CAR CRANKING))
    (OWN.VALUE GAS.TANK ?CAR EMPTY)
  THEN
    (OWN.VALUE BATTERY ?CAR DEAD)))
```

Clearly, RULE2" is directly subsumed by RULE1. Therefore, we detect the irrelevant conditions (OWN.VALUE GAS.TANK ?CAR EMPTY) in RULE2 and (NOT (OWN.VALUE GAS.TANK ?CAR EMPTY)) in RULE1.

CHAPTER 6

LOGIC CHECKER

6.1 RULE-INCONSISTENCY DETECTION

All of the material discussed in *4.5.2 Implication Redundancy Checking via a Restricted Approach* is related to the material in this section, and we recommend reading the implication redundancy section first in order to better understand rule-inconsistency.

As a result of the research into rule-inconsistency detection, it became obvious that a better solution existed for the redundancy module of the Structure Checker. The fruit of this research is now shared by both the inconsistency and redundancy modules.

The problem of rule-inconsistency detection is to check whether contradictory actions/conclusions can be derived as a result of the satisfaction of one of the rules in the knowledge base. This is a limited, but useful form, of syntactic inconsistency checking.

Like redundancy, each LHS of a rule in the knowledge base is used to determine a valid assumption set. Specifically those members of a LHS which do not have a direct proof via a fact lookup in the extensional fact base are assumed true. Unlike redundancy, rule-inconsistency detection does not require the skolemization of the input assumption set. This allows the variables in the input assumption set to be constrained by bindings. This is why we exclude from consideration all LHS literals which already have a proof via fact lookup. Finally, unlike redundancy detection there is no constraint forcing exclusion of the parent *rule from the inference*. Skolemization was used during redundancy checking to guarantee generality, but the rule-inconsistency check allows instances of incompatibilities to be reported.

The default incompatibilities are:

1) co-existence of a proof of the presence and absence of a literal

2) co-existence of a proof of the presence of a literal and the presence of its explicit negation

3) user-created semantic constraint rules whose RHS assert an incompatibility warning. For example:

    const1: sex(?Person,male),sex(?Person,female) → (incompatible)

% Resultant Prolog Programs

1) %default incompatibility - explicit negation
```
incompatible:- not(Goal),
               call(Goal).
```

2) %default incompatibility - negation by failure
```
incompatible:- cantfind(Goal),
               call(Goal).
```

3) %user-defined incompatibility
```
incompatible:- sex(Person,male),
               sex(Person,female).
```

Definition:
E = The set of static facts, i.e. the extensional database
I = The set of rules in the knowledge base
K = E ∪ I
R = The rule being checked for rule-inconsistency.
V = All members of rule R's LHS which don't already have a proof via E
A = All potential incompatibilities.
A2 = Potential inconsistencies directly involving a member of rule R's RHS.

6.1.1 Revisiting the Provability Cache

If the provability cache is absent, rule-inconsistency exists if $V \cup K$ entails *a2*, such that *a2* is a member of *A2*. If the provability cache is present, rule-inconsistency exists if $V \cup K$ entails *a*, such that *a* is a member of *A*. So the rule-inconsistency checker will attempt more work when the provability cache is constructed.

Consider the following example:

r61: A , B → add(not(C))
r62: A → add(D)
r63: D, B → add(C)

Since the valid input set {*A*,*B*} UNIONED with the rule-set {*r61*,*r62*,*r63*} entails proof of both add(C) and *add(not(C))*, a potential rule-inconsistency exists.

Also consider the rules:

r72: A → B,D
r73: B → C
r74: D → not(C)

Satisfaction of rule *r72* causes the incompatibility *(C,not(C))*, neither of which are on the RHS of rule *r72*. When the provability cache is present, it will be searched for conflicting literals, and a proof of the rule inconsistency will be attempted. The cache is therefore very useful, and it works well in conjunction with the backward-chaining approach. Not only does the cache provide pruning, but it also provides guidance in determining which incompatibilities to check.

To reiterate, the provability cache is a collection of sub-caches. A sub-cache is associated with each rule's LHS. As originally described, the parent rule was not allowed to participate in the inference process during creation of the sub-cache. This is too restrictive for our notion of rule-inconsistency checking, since the parent rule is allowed. Therefore, each sub-cache needs to be segmented into a lhs-sub-cache and rhs-sub-cache. The lhs-sub-cache would apply when checking Redundancy. The rhs-sub-cache contains the extra candidates that are likely to be satisfiable when the RHS of the parent rule is also allowed to be used in the inference. When checking rule-inconsistency, both the lhs and rhs sub-caches are used.

6.2 CONFLICT DETECTION

Conflict detection is the simple matter of attempting to prove from existing facts and the rules an instance of any known incompatibility. Unlike the rule-inconsistency checker, there are no assumption sets in this process.

The circuit domain KB (which is one of the example files in the test case file, see the *Software Test Description for the DARPA Expert System Validation Associate*, Appendix A.) includes an example of conflict. The example is built on top of a KB which correctly models AND and XOR gates. In this scenario, we have the results of an AND gate and an XOR gate being fed incorrectly via two connections to the same port of another AND gate. In the face of specified input values, port IN2 of gate AND2 will have both the value 0 and the value 1. The circuit KB also contains a rule indicating that it is incompatible for a port on an AND gate to have both a 0 and 1 value at the same time.

Conflict Example:

```
incompat1:    and(?Port,?Gate,1),and(?Port,?Gate,0) →
              text(incompatible).
xor2_to_and1: xor(xor2,out,?Value) →
              add( and(and1,in2,?Value)).
and2_to_and1: and(and2,out,?Value) →
              add( and(and1,in2,?Value)).
```

```
% Seed facts to cause a conflict.
% XOR2 will produce a 0.
% AND2 will produce a 1.

xor(xor2,in1,1).
xor(xor2,in2,1).
and(and2,in1,1).
and(and2,in2,1).
```

6.2.1 Handling Negation by Failure during Conflict Checking

During redundancy checking, provability of cantfind(?Literal) could only be achieved via an explicit proof that ?Literal had been deleted. If a redundancy was proven to exist, the claim would be that the redundancy would still exist regardless of the inclusion of new information.

On the other hand, the point of conflict checking is to determine incompatibilities resulting from the present state of the knowledge base; therefore, it makes sense to simulate the negation-by-failure operator (cantfind) during conflict checking. For example, the following should be an example of conflict assuming there does not exist a way to prove *C*.

Conflict Example using Negation by Failure:

```
% Rules
   r81: A → B
   r82: D, cant_find(C) → not B
% Facts
   A
   D
```

6.3 NUMERICAL COMPLETENESS CHECKER

The numerical completeness checker module, originally a part of the logic checker, is described under the Omission Checker (see section 9.4.1 Numerical Completeness Checker).

CHAPTER 7

EXTENDED STRUCTURE AND LOGIC CHECKERS

7.1 EXTENDED MODE VERSIONS

The extended versions of the Structure and Logic Checkers can be seen as a way for these checkers to correctly handle some of the more obvious aspects of frames and class inheritance (the *isa* meta-relation), as well as an extended form of unification (rewriting) expressed by synonyms for slots (hereafter referred to as the meta-relation *synonym*), and subrelationship between slots (hereafter referred to as the meta-relation *isa_rel*, or *subrelation*). *Synonym* can be useful for finding problems which can result from merging class definitions in separate KB's. The *isa_rel* meta-relation can be useful for modeling class inheritance when slots refer to instances of classes.

The extended versions of the various checkers can be used by first switching the mode in which DEVA operates from *regular* mode to *extended* mode (see *Circular Buttons* under section 3.2.2 Options Window). Since the extended versions of these checkers were implemented by changing the way these checkers access rule KB data, switching between modes is really just a switch in access methods. All of the abilities of the Structure and Logic Checkers in regular mode are found in the extended mode. When DEVA is in extended mode, the extended version of unification is available to all the checkers. When in regular mode, only classical unification without respect to the meta-relations (*isa*, *synonym*, and *isa_rel*) is used.

Using the *synonym* and *isa_rel* meta-relations requires consistent methods for applying *synonym* rules on *flat* structure checkers. Here flat means those checkers which do not emulate inferencing to perform their respective checks, but instead rely on direct matching (actually unification, which is more general) to obtain their efficiency. A reference to an *indirect* checker means that this checker emulates inferencing to arrive at its results. An example of a flat checker is *direct subsumption* or *deadend*, while an example of a non-flat checker is *implication redundancy* (also referred to as *indirect subsumption*).

7.2 EXTENDED MODE CONVENTIONS

The *synonym* meta-relation applies to frame slot names only and not to class names. Synonymy between any two slots means that they can be treated as equivalent during any unification that occurs in the flat or indirect checkers. Synonymy as applied to classes would involve checking both classes to make certain they have a common inheritance from the superclasses above them, and the same structure (slot names, number of slots, facets, etc.). The synonymy relation permits and facilitates the validation of two or more merged KBs or parts of a KB. It requires, however, that the underlying shell support the declaration of synonymy. This is currently not the case in KEE.

The *isa_rel* meta-relation, like *synonym*, applies only to frame slots, but unlike *synonym*, it is not bidirectional. For instance, if *murderer_of* is a subrelation (sub-slot) of *killer_of*, then the values for *murderer_of* slot are valid for the *killer_of* slot, but the values of *killer_of* are not necessarily valid for the *murderer_of* slot. To see why the relation does not hold in this example, a person's killer may have done it unintentionally, and thus would not qualify as a murderer. This module checks the KEE fact base for *isa_rel* relationships. Checking the KEE rule base requires that the *isa_rel* has also been defined in KEE using rules similar to

DEVA's *isa_rel* rules.

The *isa* meta-relation applies to classes and instances (members) of classes only. This is consistent with the traditional interpretation of *isa*, where KEE superclass information is held in the *isa* meta-relation. Direct class inheritance is obtained through the transitive evaluation of DEVA's *isa/2*. *Isa* between classes, *isa(C1,C2)*, means that class *C1* is a subclass of *C2* and not vice-versa.

Of course, the meta-relation for *isa* between classes (classical inheritance) does not require additional KEE rules to give KEE the properties of class inheritance, since inheritance is an integral part of the KEE inference engine. *Isa* applied to classes and tests for membership in a class are properties that arise from KEE's class representation.

7.2.1 The Difference Between Classes and Slots

It is helpful to examine DEVA's representation of KEE's classes (frames) and class slots in order to understand the output produced by DEVA and the difference between *isa* applied to classes and *isa_rel* applied between slots within a class.

There are some assumptions about DEVA's class representation that are necessary for the correct operation of the structure and logic checkers in extended mode. *Synonym* and *isa_rel* rules must deal exclusively with class slots, while *isa/2* must operate exclusively on the arguments of *member_of/2* and *isa/2* literals appearing in KB rules (the arguments for these literals are always class names and class instances).

To see what this means, assume we have the meta relation between two classes, *isa*(boy,human), a meta-relation between two slots, *synonym*(son,boy), and a class father with slot *boy* (don't confuse this with the class *boy*), indicating the father's boy, which would have as values the identifiers of instances of the class *boy*, e.g., boy1, boy2, etc. To summarize:

```
< < KEE FRAME> >
UNIT: father1
MEMBER OF: father
OWN SLOT: boy
VALUES: boy1

< < DEVA FACTS> >
member_of(father1,father)
boy(father1,boy1).

< < KEE FRAME> >
UNIT: boy1
MEMBER OF: boy
OWN SLOT: name
VALUES: bill
SUPERCLASSES: human

< < DEVA FACTS> >
member_of(boy1,boy)
name(boy1,bill).
isa(boy,human).
```

```
< < KEE RULE> >
(RULE lineage
   (IF (IN.CLASS ?X boy)
       (IN.CLASS ?Y father)
       (THE boy of ?Y is ?X)
   THEN  (THE parent of ?X is ?Y)))

< < DEVA RULE> >
rule(lineage,
     [member_of(X,boy),
     member_of(Y,father),
     boy(Y,X)],
     %= >
     [add(parent(X,Y))].
```

There exist two alternatives for representing class membership with a DEVA literal.

1) boy(X)
2) member_of(X,boy)

In the first case, we may say that the class name appears as the functor (literal name), and in the second case, the class name appears as an argument (occurring inside the literal). As we can see in the example above, DEVA uses the second representation, namely *member_of/2*.

What happens when we adopt the first representation *boy(X)*? It would be difficult to distinguish between slots and classes because slots are represented in the form: *slot_name(Object,Value)*, e.g. *boy(father1,boy1)*. The functor of an arbitrary literal in a DEVA rule could be referring to a slot name, or to a class name, as shown above. By applying the *isa_rel* meta-relation to an arbitrary literal functor, we might be applying it to a slot name or a class. When we try to apply *isa/2* to an arbitrary literal, we could be applying it to a class or to a slot.

What are the consequences if we uniformly adopt the second representation which is of the form: *member_of(X,boy)* (as DEVA indeed does)? If we have again chosen to represent slots in frames as slot_name(Instance,Value), the second class representation is consistent with this slot representation. The *isa* meta-relation can always be applied to the first argument of an arbitrary literal, which can be a class name or the name of an instance of that class (member) or a variable. Likewise, *synonym* can always be applied to an arbitrary literal's functor, because there is no clash with class names.

So the second representation of classes *member_of(boy,X) is preferable to the first boy(X) because we avoid applying isa* to slot names and *synonym* to classes. No clashing between class names and slot names will occur.

As a consequence of this representation, the first argument of any frame slot in DEVA is expected to be a class instance.

As can be seen in the above example, the name *boy* can refer to a class name or a slot name. The KB developer has decided that the values of the *boy* slot are indeed instances of the *boy* class. To model the inheritance of *isa(boy,human)*, the KB developer would use *isa_rel/2* to

specify *isa_rel(boy(X,Val),human(X,Val))*, meaning that this same directional relationship applies to slots of the with the names *boy* and *human*, because their values are indeed instances of the *boy* and *human* classes.

7.2.2 Meta-relation Rule Form

Synonym and *isa_rel* constraints are defined by the developer using the Semantics Checker constraint-rule forms for Synonymy and Subrelation respectively. The developer defines these rules as members of a special DEVA constraint KB described in section 8.1. The format of these constraint rules is more restricted than for "normal" KEE application rules. These restrictions help to identify them as semantic constraint rules, as opposed to other rules, and make explicit the fact that no actual inferencing occurs with respect to application rules in a regular KB and these special semantic constraint rules.

The exact syntax of these two types of semantic constraint rules will be described in more detail in section 8.0, on the Semantics Checker.

7.2.3 Facts and Extended Mode

In DEVA, facts present in the KB are really just solitary RHS literals rules whose LHS is always trivially true; therefore, the rewriting implied by *synonym* and *isa_rel* must apply to facts as well. So if *synonym(urgency(Instance,Value), immediacy(Instance,Value))* exists, and there also exists a fact *urgency(bill,great)* in the fact base, then there also exists, virtually, a fact *immediacy(bill,great)*.

The *isa* meta-relation between classes must also be applied to any facts which have classes or class names as their arguments. For the correct operation of the various checkers in extended mode, the checkers expect the following two bodies of facts to exist. The first type of fact is:

member_of(< instance> ,< class>),

and the other is:

isa(< subclass> ,< superclass>).

These facts represent the state of the KEE KB inheritance, and will satisfy literals of the same name that may appear in KB rules.

The *member_of/2* fact is sent from the KEE environment as a message and then asserted to the DEVA environment every time a new instance of a class is created or loaded.

The *isa/2* fact is sent from the KEE environment as a message and then asserted to the DEVA environment every time a new KEE class is declared as a subclass of another, or when the declaration is initially loaded (along with the KEE KB).

The *isa/2* fact is evaluated transitively, as would be expected. If the following *isa* facts are sent from KEE:

isa(child,person)
isa(toddler,child)

then a DEVA rule:

rule(rule_1,[isa(X,person),...],[...])

with the *isa* antecedent literal would be satisfied with

isa(child,person) and isa(toddler,person).

The *member_of/2* fact is evaluated transitively with respect to any *isa* facts as well. Using the above *isa* facts, and the additional:

member_of(billy,child)

any rule precondition *member_of(X,person)* would be satisfied by *member_of(billy,person)* even though *member_of(billy,person)* does not explicitly exist. This makes sense when one considers that a child bill can have the properties of a person as well as those of a child (possess the same member slots as a person), through class inheritance.

7.3 THE IMPLICATIONS OF USING EXTENDED MODE

The effect of using extended mode upon the various structure and logic checkers arises from the extended unification available. Assume that these meta-relations hold in the following examples:

isa(patient,person)
isa_rel(boy(I,C),human(I,C))
synonym(urgency(X,Val),immediacy(X,Val)
synonym(urgency(X,Val),necessity(X,Val)

The dead-end checker can actually report fewer dead-ends under extended mode than in regular mode. This is because a *synonym* or *isa_rel* can apply between what was formerly a dead-end literal and another literal on the RHS of another rule. Checks for membership within a class (using *member_of/2*) and queries about class inheritance (using *isa/2* as a literal in KB rules) can be satisfied by the transitive evaluation of *isa* between classes, and thus will not appear as dead-ends.

The unreachability checker under extended mode can present the user with fewer examples of unreachable literals in the RHS of rules because there may exist an LHS literal which calls upon the RHS literal examined when the extended unification is taken into account.

More examples of subsumption (direct pairwise and indirect) are possible in extended mode, while less examples of ambiguity between two rules result since rewriting between slots may qualify the pair of rules being inspected in the ambiguity checker as having a subsuming relationship instead. The following rule *r1* would be ambiguous with *r2* in regular mode, but would subsume *r2* in extended mode:

r1: member_of(X,patient), near_death(X,true) → add(immediacy(X,great))
r2: member_of(X,patient), near_death(X,true) → add(urgency(X,great))

With respect to *isa* between classes, rules of the form *r3* could subsume those in form *r4* only

in extended mode:

r3: member_of(X,person),A → B
r4: member_of(X,patient),A → B

If *isa(patient,person)* then rules similar to r3 which deal with persons are more general than rules r4 that deal with patients.

The possible cycles reported by the cycle checker increase when in extended mode. The following rule shows a direct cycle which would appear in extended mode but not regular mode if *synonym(immediacy(X,Y),necessity(X,Y)* is true:

r5: member_of(X,patient), immediacy(X,great) → add(necessity(X,great)).

There are more cases of irrelevancy (direct and indirect), conflict, and inconsistency in the respective checkers because of the many possible rewrites allowed in extended mode.

Whenever rewrites can be practically expressed to justify the existence of a condition which the various checkers look for, an associated justification, or trace, of the rewriting is displayed. It is impractical to show such a justification for the unreachability and dead-end checkers, since there may exist a myriad number of justifications explaining why a literal is unreachable or a dead-end.

7.4 IMPLEMENTATION METHODS

The evaluation of the *synonym* meta-relation is done in two steps. When the synonym is added to the DEVA environment, a synonym set is computed. Then during run-time checking, a two-level expression of the synonym set is used to obtain an efficient rewriting between slots.

This synonym set consists of all slots which have been determined to be equivalent. The equivalency is determined transitively and reflexively. So if the slot *father* is synonymous with the slot *padre* in one synonym rule, *padre* is synonymous with *dad* in another, and *father* is synonymous with *poppa*, the synonym set is {*father,padre,dad,poppa*}. The *father* and *dad* slots are synonyms even though this is not explicitly stated in any synonym rule. This limited form of preprocessing saves time when compared to determining the mapping of a slot to all other synonymous sets during run-time checks for synonymy.

The construction of synonym sets is done by first collecting all the distinct slot nodes mentioned in any synonym rules. Then each of the nodes is examined. If the node is already part of a synonym set, the algorithm goes on to the next node. If it is not included, the bidirectional graph of synonym relations are searched in a breadth-first fashion. Circular, non-ending paths expressed by the synonym rules are avoided by maintaining a list of visited nodes during the search. When a node has been previously visited, the search tries another alternative. So this search which determines the synonym set is guaranteed to terminate.

The computation of synonym sets from KEE synonym rules before using them in the checkers serves to flatten any search for synonyms into a two-level mapping. This is an obvious efficiency gain over a search which could be N-1 levels deep in the worse case, where N is the number of synonym rules present in the system.

The data structure used to represent synonyms is not a simple list. For each set, a unique literal is generated which respects the bindings of the slots in the set, i.e. it respects the class instances and slot values referred to in the synonym rules. Then this unique literal, appearing nowhere else in the entire KB, is used to map each of the components of the synonym set to all other members in the set, including itself (since a slot in the set is trivially a synonym with itself). The data structure which maps a slot down to the unique literal is *syn_fan_in(Slot,Unique_Slot)*. The data structure which maps the unique literal back out to all the slots in the same set is *syn_fan_out(Unique_Slot,Slot)*. One can imagine the data structures working together graphically as an hour-glass shape, with the narrowing at the center by the unique literal.

There exists another meta-relation which maps a slot to another transitively, but not reflexively (i.e. in one direction only). This is the *isa_rel* meta-relation. When an *isa_rel* meta-relation is added to the DEVA environment, it is evaluated in two steps.

The first step rectifies the *isa_rel* meta-relations with any *synonym* meta-relations. Since synonymy between slots and *isa_rel* both deal with slots, there are times when a *synonym* rewrite can apply to one or more of the slots involved in an *isa_rel* meta-relation. If one considers the *isa_rel* meta-relation as a directed relation, then a collection of chained *isa_rel* meta-relations can be viewed as a directed graph, a tree with the slot names as nodes and the *isa_rel* relation as arcs. If *synonym* rewrites apply to some of the nodes, the tree can become quite 'bushy' with many subtrees being added at these nodes.

The important thing to note is that all of these subtrees being added are exactly the same as their sibling subtrees. For this reason, the *isa_rel* facts are examined to find the *synonym* nodes. These nodes are replaced with the unique literal which represents the *synonym* sets evaluated previously, collapsing the redundant subtrees into one equivalent subtree and lessening the time it takes to search the tree during run-time checking.

The recursive search of *isa_rel* facts during run-time never returns one of these unique literals as its result. When these unique leaf nodes are reached, the unique literal is mapped back out to all of the elements of the synonym set it represents.

Finally, when an *isa_rel* meta-relation is added to DEVA, all the *isa_rel* facts in the DEVA environment are searched as though they were the starting point of a tree, to make certain that the graph(s) thus represented are indeed acyclical. This must be done because a recursive evaluation is done in run-time checkers, and cycles would introduce infinite recursion. Any cycles that appear with the addition of *isa_rel* meta-relations are displayed via the DEVA interface, and a flag is set which turns off the evaluation of the *isa_rel* meta-relation in the run-time checkers. Any checking will occur without the benefit of *isa_rel* mappings until the cycles are removed by the user.

CHAPTER 8

SEMANTICS CHECKER

8.1 SEMANTIC CONSTRAINTS RULE FORMAT

A Semantic Constraints KB is created by the KEE translator automatically whenever an application KB is created, or loaded (see section 4.1). The KB-constraint contains DEVA information, i.e., semantic or metainformation, in the form of KEE rules about the application KB. We selected the KEE syntax for these constraint rules to permit the developer to make DEVA statements without having to learn a new language.

Constraint rules may be one of two forms depending on the type of constraint, success driven or failure driven. While both types of rules have a LHS and a RHS, the LHS's and RHS's are used differently.

8.1.1 Success Driven Constraint Rules

Success driven constraint rules are rules where the LHS of the rule is used to define the context to which the constraint applies, and the RHS defines the constraint. As an example take the rule *SC1*, which states that if someone is the father of a child, then they must be members of the same class, the father must be male, and the father must also be the parent of the child. (To facilitate the discussion of some points, we have commented each potentially relevant line with *;n.m*, where n is the rule number and m, the line number.)

```
(SC1
   (IF    (THE FATHER OF ?CHILD IS ?DAD)  ;1.1
    THEN
          (THE SEX OF ?DAD IS MALE)  ;1.2
          (?CHILD IS IN ?SPECIES)  ;1.3
          (?DAD IS IN ?SPECIES)  ;1.4
          (THE PARENT OF ?CHILD IS ?DAD)))  ;1.5
```

The more general the LHS, the more situations exist where the constraint may be applied. Only when the LHS conditions are satisfied, do the RHS constraints apply. Alternatively the rule *SC1* could have been written as rule *SC2*, which states that if someone is the father of a child, and they are members of the same class, then the father must be male, and the parent of the child.

```
(SC2
   (IF    (THE FATHER OF ?CHILD IS ?DAD)  ;2.1
          (?DAD IS IN ?SPECIES)  ;2.2
          (?CHILD IS IN ?SPECIES)  ;2.3
    THEN
          (THE SEX OF ?DAD IS MALE)  ;2.4
          (THE PARENT OF ?CHILD IS ?DAD)))  ;2.5
```

These two rules differ in the conditions under which the constraint applies and in the constraint itself. Take the example of *father_of(fluffy,fido)* where *member_of(fluffy,cat)* and *member_of(fido,dog)*. The rule *SC1*, would generate a warning message that the father of *fluffy* the *cat* is a *dog*, thus violating the class membership constraints (RHS lines 1.3, 1.4). The

rule *SC2* would generate no warning because the LHS would not be satisfied. The conditions dealing with class membership (LHS lines 2.2, 2.3) do not apply to *father_of(fluffy,fido)*.

Examples of success driven constraints are *min-max-set*, *inverse*, *subrelation*, *value_set* and *value_range*. All of these semantic constraints are described in detail later in this section.

8.1.2 Failure driven constraint rules

Failure driven constraint rules are rules where the LHS of the rule is used to define a context in which a constraint has been violated, and the RHS names the type of constraint. As an example take the rule *SC3*, which states that if someone is a member of the class *person* and their *gender* is known to be both *male* and *female*, then this is incompatible.

```
(SC3
   (IF    (?X IS IN PERSON)  ;3.1
          (THE GENDER OF ?X IS MALE)  ;3.2
          (THE GENDER OF ?X IS FEMALE)  ;3.3
    THEN
          (INCOMPATIBLE)))  ;3.4
```

Failure driven constraints work by attempting to prove the goal on the RHS (line 3.4), based on the current state of the knowledge base.

Examples of failure driven constraints are *incompatible*, *subrelation*, and *interstate_integrity*.

8.2 INVERSE CHECKER

The Inverse Checker, using the semantic constraints which define the inverse property of relations, notifies the developer when these constraints have been violated based on the current state of the fact base.

Rule *SC4* is an example of an inverse constraint that states that the relations *father* and *has-father* are are inverses of one another.

```
(SC4
   (IF  (THE FATHER OF ?X IS ?Y)  ;4.1
    THEN
        (INVERSE (FATHER ?X ?Y) (HAS-FATHER ?Y ?X))))  ;4.2
```

The Inverse Checker works by finding all facts in the KB that satisfy the LHS of rule *SC4*. When a LHS matches, the checker attempts to prove the corresponding RHS subgoal, in this case *(has-father ?Y ?X)*. If the subgoal fails, a warning message is issued to the user.

8.2.1 Inverse Relation Rule Format

As mentioned above, the LHS defines the condition for which the constraint applies, and the RHS defines the constraint. The LHS is in Tell and Ask format, just like any other KEE rule. The RHS literal establishes the inverse property of the two slots. The format is

```
(inverse (slot1 Var1 Var2)  (slot2 Var2 Var1))
```

where *slot1* is the inverse of *slot2*. Notice that the arguments *(Var1 and Var2)* appear in opposite order for the two slots. At this time, Lisp expressions are not allowed in an inverse constraint rule. If a Lisp expression is used, the Inverse module naively assumes that the evaluation of the expression fails.

Note that while in KEE introducing a variable in the RHS of a rule, as in line 4.2 of rule *SC4*, generates an error message; DEVA constraints allows this.

8.3 SUBRELATION CHECKER

The Subrelation Checker takes the semantic constraints defining the relation hierarchy in particular domain and notifies the developer when these constraints have been violated based on the current state of the fact KB. The subrelation constraints (i.e., *isa_rel*) also play an important role in the Extended Mode Checkers, see section 7.1.

Rule *SC5* is an example of a subrelation constraint that states the relation *father-of* is a subrelation of *parent-of*. In other words, if Harry is the father of Sue, then it is the more general case that Harry is also the parent of Sue. The rule also places certain data type constraints on the arguments to the relations.

```
(SC5
   (IF    (THE FATHER OF ?X IS ?Y) ;5.1
    THEN
          (THE SEX OF ?X IS MALE) ;5.2
          (?X IS IN ?Z) ;5.3
          (?Y IS IN ?Z) ;5.4
          (SUBRELATION (FATHER ?X ?Y) (PARENT ?X ?Y)))) ;5.5
```

The Subrelation Checker works in much the same way as the Inverse Checker, by matching facts in the KB with the LHS of the constraint rule. When a fact matches the LHS, the Subrelation Checker attempts to prove each corresponding subgoal on the RHS. If any one subgoal fails, a warning message is issued to the user.

8.3.1 Subrelation Constraint Rule Format

The format for a subrelation is very similar to that for an inverse. The LHS is in Tell and Ask format, just like any other KEE rule. The RHS can be divided into two parts. The first part, which is used to establish data type constraints on the relation arguments, is also in Tell and Ask format (lines 5.2 - 5.4). The second part of the RHS of the constraint rule defines the relationship hierarchy (line 5.5) between the two slots. The format is

```
(subrelation (slot1 Var1 Var2)  (slot2 Var1 Var2))
```

where *slot1* is defined as a subrelation of *slot2*. Notice that the *Var1* and *Var2* appear in the same order between *slot1* and *slot2*, unlike the inverse constraint format. At this time, Lisp expressions are not allowed in a subrelation constraint rule. If a Lisp expression is used, the Subrelation Checker naively assumes that the evaluation of the expression fails.

8.4 MINIMUM/MAXIMUM CARDINALITY CHECKER

The Minimum/Maximum Cardinality (Min-Max-Set) Checker uses semantic constraints defining the minimum/maximum cardinality of particular slots, frames, and slot values. It notifies the developer when these constraints have been violated based on the current state of the fact KB. There are four cases of Min-Max-Set: *min-max-role*, *min-max-relation*, *min-max-frame*, and *min-max-inverse_role*. Each constraint type takes a slightly different form.

This is the general RHS form for stating a min-max constraint:

(min_max_Type UnitName Slot-SlotValue-Pair Restrictions MIN-MAX)

UnitName --
{Unit | Var | NIL} where Unit could be a specific class, such as PEOPLE, or a variable bound on the LHS of the constraint, or NIL, which means that the constraint applies to all units.

Slot-SlotValue-Pair ({(Slot {SlotValue | NIL}) | NIL)}) --
A list of one or more (slot slotValue) pairs where slot is defined for Unit, and slotValue is either a specific value, or NIL, which means that any value will do. These are all valid examples of Slot-SlotValue-Pair:
((age NIL)) -- All Units that have any value for AGE
((age 39)) -- All Units whose AGE slot has a value of 39
((hair blond)(eyes blue)(age 2)) -- All Units who have blond HAIR, blue EYES and an AGE of 2.

NIL --
All Units.

Restrictions --
A list of classes of which the value of Slot-SlotValue-Pair must be a member.

MIN-MAX --
The minimum and maximum number of values allowed, where min is restricted to non-negative integers.

8.4.1 Min-Max-Role

In the min-max-role constraint, for a particular slot, restrictions or qualifications can be placed on that slot and the number of slot-values that satisfy those conditions are counted.

Example of the general form:

```
(Rule-Name
   (IF    {LHS | NIL}  ;6.1
    THEN
         (MIN_MAX_ROLE UnitName  ;6.2
         Slot-SlotValue-Pairs  ;6.3
         ClassRestrictions  ;6.4
         Min-max  ;6.5
```

In line 6.2, *UnitName* may be either a Class name, a variable bound on the LHS of the constraint, or NIL. A set is constructed for each *Slot-SlotValue-Pair* defined in line 6.3 where the second element in *Slot-SlotValue-Pair* is a member of the classes in *ClassesRestrictions* in line 6.4. Line 6.5 defines the lower and upper cardinality of the elements in the union of

these sets.

The following is an example of min-max-role:

```
(FRESHMAN_ELECTIVES
   (IF    (?P IS IN PERSON)  ;7.1
          (THE STATUS OF ?P IS FRESHMAN)  ;7.2
    THEN
          (MIN_MAX_ROLE ?P   ; 7.3
                ((ENROLLED NIL))  ;7.4
                (ELECTIVES)  ;7.5
                (0 1)))))    ;7.6
```

This min-max-role constraint states that a freshman may be enrolled in no more than one elective.

Line 7.3 is an example of a variable bound on the LHS of the constraint. If, on line 7.3, *?P* had been replaced by PERSON or NIL, then the constraint would have limited the number of electives anyone could enroll in, not just freshmen.

Line 7.4 defines the *Slot-SlotValue* pair on which the constraint is placed. min-max-role is unique in that only one *Slot-SlotValue* pair is allowed. If the *SlotValue* is NIL, then a set is constructed containing the number of slot values.

Line 7.5 defines the data type restrictions on the *SlotValue* pair.

The constraint FRESHMAN_ELECTIVES queries the database for any one person who has a value of FRESHMAN for the slot STATUS (lines 7.1 and 7.2). Lines 7.4 and 7.5 construct a set of all *?P's* slot values for ENROLLED that are members of the class ELECTIVES. Line 7.6 states that there may not be more than 1 nor less than 0 members in that set.

8.4.2 Min-Max-Frame

The min-max-frame constraint counts the number of Units that have at least one slot value that satisfies the restriction or qualification placed on that slot. Multiple slot values are counted as one.

An example of the general form:

```
((Rule-Name
  (IF ({LHS | NIL}) THEN
     (MIN_MAX_FRAME UnitName
               Slot-SlotValue-Pairs
                  Restriction
                  Min-max)))
```

The syntax of *min-max-frame* differs from *min-max-role* in that the number of Slot-SlotValue-Pairs is not limited to one. A set is constructed using from the database matching of any number of Slot-SlotValue-Pairs, and failure to match on anything does not negate the process.

An example of the general form:

```
((NUM_ENROLLED_IN_ALL_CLASS
   (IF    (?CLASS IS IN COURSES)  ;8.1
    THEN
       (MIN_MAX_FRAME PERSON
                ((ENROLLED ?CLASS)(AUDITING ?CLASS))  ;8.2
                NIL
                (100 3000))))
```

The constraint NUM_ENROLLED_IN_ALL_CLASS states that the number of students enrolled in, or auditing one or more courses may be no less than 100, nor more than 3000. Notice that ?CLASS is bound on the LHS side (line 8.1) of the constraint and used on the RHS, (line 8.2). In addition, notice that there are two Slot-SlotValue-Pairs (line 8.2).

8.4.3 Min-Max-Relation

In the min-max-relation constraint the checker counts the number of Units that have at least one slot value which satisfies the restriction or qualification placed on that slot. Multiple slot values are counted as unique values.

An example of the general form:

```
((Rule-Name
  (IF ({LHS | NIL}) THEN
     (MIN_MAX_RELATION UnitName
              Slot-SlotValue-Pairs
                 Restriction
                 Min-max)))
```

The syntax for *min-max-relation* is the same as *min-max-frame*. Functionally, min-max-frame and min-max-relation differ only in that the former counts one or more slot values as one, and the latter computes the number of slot values in the KB. For example:

```
(TOTAL_STUDENTS
  (IF NIL THEN
     (MIN_MAX_RELATION PERSON
              ((ENROLLED NIL)(AUDITING NIL))  ;9.1
              NIL
              (500 10000))))
```

The constraint TOTAL_STUDENTS states that the number of classes the student body as a whole is collectively enrolled in or auditing may be no less than 500, nor more than 10000. Notice that NIL is used in both *Slot-SlotValue-Pairs* in the constraint rule. This rule constrains the total number of tuples (records) for ENROLLED and AUDITING.

8.4.4 Min-Max-Inverse-Role

The min-mix-inverse-role constraint counts the number of Units that have a particular value for a slot.

The general form:

```
((Rule-Name
  (IF ({LHS | NIL}) THEN
    (MIN_MAX_INVERSE_ROLE
        UnitName
      Slot-SlotValue-Pair
       Min-Max))))
```

Notice that the parameter list is different from any of the other min_max constraints. The *Restriction* parameter has been removed. *UnitName* and *Slot-SlotValue-Pair* are the same as for each of the other types of min-max constraints. For example:

```
(UPPER_DIVISION_CLASS_SIZE
  (IF (?CLASS IS IN UPPER_DIVISION) THEN
    (MIN_MAX_INVERSE_ROLE PERSON
                ((ENROLLED ?CLASS))
                (5 12))))
```

The constraint UPPER_DIVISION_CLASS_SIZE places a limit on the number of students enrolled in an upper division class to no less than 5, no more than 12.

```
(ENROLLED_TO_AUDIT_RATION
  (IF (?CLASS IS IN COURSES)
     (THE ENROLLED OF ?CLASS IS ?E)   ;11.1
   THEN
   (MIN_MAX_INVERSE_ROLE PERSON
                ((AUDITING ?CLASS))
                (0 ?E))))          11.2
```

The constraint ENROLLED_TO_AUDIT_RATION says that the number of students auditing a class may not exceed the number of students enrolled in a course. Notice the use of the variable ?E in MIN-MAX (line 11.2), that was bound on the LHS (line 11.1) of the rule.

8.5 INTERSTATE INTEGRITY CHECKER

The Interstate Integrity Checker takes constraints written by the knowledge base developer and notifies the user when these constraints can be violated based on the current state of the knowledge base (rules plus facts). For KEE-based systems, constraints take the following form:

```
IF (body of constraint)
THEN
(INTERSTATE_INCONSISTENCY)
```

The following is an example of an interstate integrity constraint that states that a person's salary may not decrease:

```
IF  (?P IS IN PERSON)
    (SALARY ?P ?SALARY1 ?YEAR1)
    (SALARY ?P ?SALARY2 ?YEAR2)
    (LISP (<  ?YEAR1 ?YEAR2))
    (LISP (<  ?SALARY2 ?SALARY1))
THEN
    (INTERSTATE_INCONSISTENCY)
```

The Interstate Consistency Checker works by attempting to prove the goal *INTERSTATE_INCONSISTENCY* based on the current state of the knowledge base. Because such a computation may take an arbitrarily long time, this checker uses a user-supplied depth bound to limit the amount of chaining it will undertake. In addition, the user has the ability to interrupt this processing in order to undertake other tasks.

Errors may be introduced into the proof process through the use of Lisp antecedents. DEVA recognizes a limited number of lisp expressions (primarily simple arithmetic expressions), and attempts to evaluate them. When DEVA does not recognize a lisp expression, it naively assumes that the evaluation returned successfully.

8.6 INCOMPATIBILITY CHECKER

The Incompatibility Checker is a failure driven constraint as described in Section 8.1.2. The Incompatible Constraints are used by the Semantics Checker to check the fact KB for inconsistencies. They are also, more importantly, used by the Logic Checker to check the rule base for consistency, see Sections 6.1 and 6.2.

The following DEVA rule, which defines an Incompatibility Constraint dealing with the the rotation speed of airplanes, may serve as an example of the DEVA "KEE-syntax" format:

```
(ROTATION-SPEED
   (IF     (?X IS IN PLANE)
           (THE SPEED OF ?X IS ?SPEED)
           (THE STATUS OF ?X IS ROTATION-TAKEOFF)
           (LISP ( <  ?SPEED MINIMUM-ROTATION-SPEED))
   THEN
           (INCOMPATIBLE)))
```

Here the system developer is stating that it is incompatible for an airplane to perform a takeoff rotation unless its speed is greater than or equal to the minimum rotation speed allowed by flight regulations.

8.7 VALUE CHECKER

The Value Checker takes the semantic constraints defining the range or set a slot value may have in a particular domain and notifies the de eloper when these constraints have been violated based on the current state of the fact KB.

Rule SC6 is an example of a value constraint that states that a *fighter_jet* may only have a slot value for *mission* of [bomb_run, refuel, air_combat]. In other words if fighter_jet *St. Louis* had a mission of *transport* this constraint would be violated.

```
(SC6
   (IF    (?P IS IN FIGHTER_JET)
          (THE MISSION OF ?P IS ?M)
    THEN
          (VALUE_SET ?M (BOMB_RUN, REFUEL, AIR_COMBAT))))
```

The Value Checker works by matching facts in the KB with the LHS of the constraint rule and then testing the value of a constraint specified variable. If that variable is not within its specified range or set a warning message is issued to the user.

8.7.1 Value Constraint Rule Format

The LHS of the Value Constraint is in Tell and Ask format, just like any other KEE rule. The constraint RHS is divided into three parts: value_constraint_type, constraint_variable, and value_constraint. Value_constraint_type (either value_range or value_set) specifies whether the constraint is covered by a range or set constraint. Constraint_variable specifies the variable that was matched on the LHS of the constraint, that the developer wishes to constrain. Value_constraint is either a range or set depending on the value_constraint_type.

A range constraint takes the form (low high), where *low* is a number or type symbol ni_, standing for Negative Infinity; where *high* is a number or the symbol pi_, standing for Positive Infinity.

A set constraint takes the form (set) or named set where set is a list of legal values, or a named set which is one of the following: [is_digit,is_alpha,is_ascii,is_alnum].

is_digit: 0..9
is_alpha: a..z, A..Z
is_ascii: any ascii code from 0 to 127
is_alnum: any alpha-numeric a..z,A..Z,0..9

General form:
(value_constraint_type constraint_variable value_constraint)

Examples:
(value_range ?age (0 100))
(value_range ?age is_digit)
(value_set ?gender (male female boy girl))

In each of the examples the variable (ie ?age) must be set on the LHS of the constraint.

CHAPTER 9

OMISSION CHECKER

9.1 INTRODUCTION

One can model much of the knowledge in knowledge-based systems around the concept of sets, e.g., a set of classes of objects, a set of relations, a set of rules, and a set of values for a multi-valued slot. Given a set written by the developer, the basic question to ask is "Is the set complete?". In other words, does the set contain all the necessary elements or does it miss some elements? A closely related question is "Given a set, do the explicitly mentioned subsets completely cover this set?". The goal of completeness checking is to answer these questions by investigating and identifying useful techniques and representations for defining completeness.

9.2 OMISSION OF CLASSES

In KEE, as in most frame-based systems, the frames are organized in a hierarchy of classes, where generality increases as one approaches the top of the hierarchy. Frames are defined in terms of their slots or attributes, so a subclass may specialize a class by having additional slots defined, or by having additional restrictions placed on a slot (e.g. by restricting the number of allowable values), or both.

DEVA checks for completeness of the class hierarchy in two ways; the first uses rules defined in the meta-KB to indicate those slots that are most meaningful in the development of subclasses; the second infers *potentially meaningful* subclasses based on patterns in rules present in the regular KB.

9.2.1 MetaKB-Based

DEVA allows the knowledge engineer to provide an object model in the meta-knowledge base that provides guidelines for subdividing classes. For example, a metarule may say that the class *Person* can be grouped by the slots *sex* and *age*. If the developer creates only the classes for *Man*, *Woman*, and *Boy*, this omission checker will recommend adding a class corresponding to *Girl*.

The metarules used by the class omission checker take the following KEE form:

```
IF  (< ?Var1>  IS IN < Class_Name> )
    (THE < Slot_Name>  OF < ?Var1>  IS < ?Var2> )
    (other conditions)
THEN (SUBCLASSING_RULE)
```

The following is the KEE representation of the subclassing rules for *Person* given above:

```
IF (?P IS IN PERSON)
   (THE AGE OF ?P IS ?A)
THEN (SUBCLASSING_RULE)

IF (?P IS IN PERSON)
   (THE SEX OF ?P IS ?S)
THEN (SUBCLASSING_RULE)
```

At this time, the class omission checker ignores the *other conditions* part of the metarule, so the class omission checker operates based on class and slot names alone. The *other conditions* may be used in future, enhanced versions of this checker.

The class omission checker generates subclasses based on both the minimum and maximum cardinalities and the value classes of slots specified in the metarules. It currently handles value classes which are defined to be one of a set of values, intervals of integers, or user-defined classes.

The class omission checker may suggest more than one level of subclasses beneath the class specified in a metarule; i.e, it suggests the addition of a new subclass hierarchy in the form of a potentially complex directed acyclic graph. This graph retains the original subclasses, since presumably these are important to the knowledge engineer. However, because these original classes may inadequately subdivide the original parent class, it may also include suggested classes which serve not only to cover, but also to partition, the original parent.

As an example, suppose the class *Person* had the immediate subclasses *Female_Voters* (female, age > 17) and *Male_Illegal_Drinkers* (male, age < 21) and a meta-knowledge base constraint stating that *Person* can be subdivided by the slot *age*. The omission checker suggests retaining the original class structure, with the addition of three new subclasses that would partition *Person* based on age, that is, (age < 18), (age 18-20), and (age > 20). If there were an additional meta-KB constraint that stated that *Person* could be subdivided based on the slot *sex* as well, this checker would recommend subclasses corresponding to (male), (female), (male, age < 21), (female, age < 21), (male, age > 17), (female, age > 17), (male, age < 18), (female, age < 18), (male, age 18-20), (female, age 18-20), etc.

The recommendations include both pre-existing subclasses and new subclass suggestions. *Male_Illegal_Drinkers* remains in the hierarchy, but its female counterpart, (female, age < 21), is also present. These subclasses have subclasses of their own, e.g., (male, age < 18) is a subclass of *Male_Illegal_Drinkers*.

In essence, the graph produced is the cross-product of the recommendations based upon individual slots. Because the number of recommendations generated via the cross-product rapidly becomes unwieldy, DEVA gives the user the option of viewing only the single slot recommendations or of viewing the cross-product of selected slots when a large graph would result.

9.2.2 Rule-Based

KEE is a system that allows a knowledge engineer to encode knowledge using both frames and rules. Although information stored in one form may easily be transformed into the other, frames are generally more useful for storing terminological knowledge, whereas rules are more useful for storing heuristic knowledge. KEE performs some semantic and

consistency checks on frames, but does little checking of rules. Thus, it is often useful to transfer knowledge from rules to frames based on patterns occurring in rules. Where applicable, such a transfer makes the knowledge base more structured, hence more amenable to easy debugging.

The rule-based omission checker proposes new rules based on two types of information: explicit DEVA *isa* metarules and recognition of patterns in object base (KEE) rules.

Often, during the development of large knowledge bases, people write rules to cover certain situations, but omit the rules that cover similar but distinct situations. If a rule exists that checks membership in one subclass of a parent class, DEVA checks to see whether there are additional rules of the same form that cover the other subclasses of the parent class.

As an example, consider the class hierarchy:

```
isa(B,A)
isa(C,A)
isa(D,B)
isa(E,B)
isa(E,C)
isa(F,C)
```

If there is a rule,

```
IF (?X IS IN  B) (THE SOME-PROPERTY OF ?X IS ?Y)
THEN
(DO_SOMETHING)
```

DEVA checks whether there is another rule in the knowledge base with the same form, e.g.

```
IF (?X IS IN  C) (THE SOME-PROPERTY OF ?X IS ?Y)
THEN
(DO_SOMETHING_POSSIBLY_DIFFERENT)
```

Notice that these two rules do not necessarily cover all individuals of *A*; there may be individuals that belong directly to *A*, that are in neither *B* nor *C*. Therefore, DEVA also checks for a rule:

```
IF (?X IS IN  A) (THE SOME-PROPERTY OF ?X IS ?Y)
THEN
(DO_ANOTHER_SOMETHING_POSSIBLY_DIFFERENT)
```

DEVA is also alert for situations involving multiple inheritance. If some other rule mentions class *E*, DEVA checks for similar rules mentioning classes *B*, *C*, *D*, and *F*, and report those classes which are not covered.

In addition to suggesting new subclasses based on rules in the meta-KB, DEVA also recommends new subclasses based on patterns in rules and based on novel combinations of existing subclasses' slot definitions. For example, if there were a rule whose conditions included:

(if (?a is in animal) (the skin_covering of ?a is fur) ...

the class omission checker inquires whether it may be useful to explicitly subdivide the *Animal* class into subclasses based on the *skin_covering* slot rather than having it implicitly subdivided via the rules. Currently, suggestions are made for rules mentioning classes having slot values that are either constant or that involve a numerical comparison.

9.3 OMISSION OF RELATIONS

This checker looks for missing subrelations of a relation. We define a relation as a unit which uses user-defined unit names as value classes for more than one slot. Because we define a relation as being a certain type of class, the relation omission checker is a specialization of the class omission checker, with the following exception: the class omission checker currently operates based on rules in the metaKB; the relation omission checker discovers relations in the object knowledge base and reports on them.

As an example, the class *Parent_Of* may be a relation with two slots, *elder* and *younger*, both of which have a value class of *Person*. *Parent_Of* may have a subclass (subrelation) *Father_Of*, with the value class of *elder* being *Man*. If *Person* has two subclasses, *Man* and *Woman*, the relation taxonomy omission checker will check to see whether a descendent of *Parent_Of* has an *elder* slot with the value class of *Woman*, corresponding to *Mother_Of*. If such a relation does not exist, DEVA will alert the user.

As does the class omission checker, the relation omission checker generates a potentially complex directed acyclic graph. The root of this graph is the original relation and the remainder of the graph consists of existing and/or suggested subrelations. Again, the graph produced is the cross-product of the recommendations based upon individual slots, so DEVA allows the user to select an abbreviated version for display.

9.4 OMISSION OF RULES

As the number of rules in a KBS increases, so does the difficulty of maintaining these rules and understanding the interactions between them. KEE provides help with this rule management problem via *rule classes*, which are intended to group rules according to their usage. KEE does not enforce any rigorous relationship between rule classes and classes; any structure that exists is that imposed by the knowledge engineer.

DEVA helps impose structure on rule classes through the rule omission checker which works by determining which classes are covered by rule classes, and comparing the coverages of these rule classes. The idea of checking omissions in a rule set is to find the most general class that is covered by that rule set. If two rule sets cover classes that are related by the subclass relationship, then the rule set associated with the smaller class may be incomplete.

For example, suppose we have a class hierarchy with a parent class *P* that has two children, *C1* and *C2*. If Rule_Class_1 has rules that cover all children of *P*, while Rule_Class_2 has rules that cover all children of *C1*, this omission checker will inquire whether Rule_Class_2 ought to have rules dealing with *C2* as well.

9.5 INCOMPLETE SLOT VALUES

In KEE, each slot of a unit has a minimum and maximum number of values that may exist

for that slot. The incomplete slot value checker checks to see that units have their slots filled with the appropriate number of values, e.g., if a man should have exactly one wife, this checker will issue a warning for those men who do not have wives (or have more than one wife). KEE will do a similar check itself if the appropriate flags are set. To check more intricate relationships on slot values, use *min_max_role* constraints as described in section 8 on the semantics checker.

CHAPTER 10

RULE REFINER

"The goal of the rule refiner is to help the developer refine his rules. Since this is an interactive process a good and comprehensive user interface is required and will be provided" -- DEVA proposal

10.1 THE RULE REFINING PROBLEM

In the traditional approach to rule refinement, an expert creates a rule, applies it to the knowledge base, observes what happens, then modifies the rule contingent on that observation. This process is repeated until the expert is reasonably satisfied that the rule reflects its creators intent.

10.2 THE DEVA RULE REFINER APPROACH

The Rule Refiner is a graphical software tool designed to assist system developers in constructing a rule based expert system. The Rule Refiner facilitates building rule based application systems that enhance the productivity and capabilities of technical experts and developers.

The Rule Refiner simulates the rule application process by showing the effects of firing a rule on a graphical model of the fact base domain called a *Unit Graph.* A Unit Graph is a dynamic undirected acyclic graph that presents the relational hierarchy of class, subclass, instance, and *Temporary Unit* links in the knowledge base. A Temporary Unit (TU) is an artificially created instance of a class, created by the Rule Refiner. One may examine the slots and slot values for all units in the unit graph (both temporary and original) to determine their relevancy to the rules.

Units (both temporary and original) that application of the rule might affect are highlighted. Any manipulation of the Unit Graph may indicate that the scope of the rule needs modification. When one or more highlighted Units are changed by the expert to an unhighlighted state, it indicates that the rule might be too general. Alternatively, if one or more unhighlighted Units is changed to a highlighted state, it might be an indication that the rule is too restrictive.

10.3 THE RULE REFINER INTERFACE

The Rule Refiner Interface consists of four windows joined together in a frame, known as the *Rule Refiner Frame.* The windows are (from upper left to lower right):

1) Rule Refiner dialog box contains:

   a) The name of the rule under refinement.

   b) Quit button that removes the unit graph.

   c) Close button that iconifies the unit graph

d) (Lisp:Prolog) circular button that controls which language the rule is displayed in the Rule Text View.

e) (Class:LHS) circular button that controls the relationship displayed in the unit graph window. When this circular button displays *Class* the unit graph displays all the Class-instance links. When it displays *LHS*, sets of units that make up one firing of the LHS of the current rule are displayed. These sets of units are connected by a line of unique color, and a like colored box is drawn around each unit of the set. Up to eight separate relations may be shown at one time.

2) Rule Text View
This view contains the text of the rule under refinement, controlled by the circular (Lisp:Prolog) button in the Rule Refiner dialog box.

3) Output View
This view contains a textual list of the algorithms used in construction of Temporary Units. A complete description of the algorithms is presented in section 5.3.

4) Unit Graph
The unit graph, as described above, lets the user see when the RHS of a rule will be applied by generating Temporary Units from available data sources that are based on the literals of the rule's LHS. Though a Temporary Unit is created and displayed in the unit graph, no new information is added to the fact base, and the creation of any TU (and the slot:slotvalues associated with it), will not affect any other checker whatsoever.

The Unit Graph Window
Class-units are light blue; member-units are either light-grey (unhighlighted) or yellow (highlighted); the background is light grey; the lines that represent the relation between a class-unit and its subclass-unit are solid scarlet, and the line that represents the relationship between a class-unit and its member-units is a dashed with a bold font. A left click on a member-unit toggles it's status (yellow = highlighted, light-grey = unhighlighted). A left click on a class-unit toggles all of its members (including member-units of its subclasses). A middle click and drag on any unit allows the user to reposition the unit within the graph.

A right-click on a class-unit displays the slots that are defined at that class in a popup menu in which the menu-items are slot names. When sub-menus (also known as walking menus) appear in the popup menu, the items that appear in a sub-menu are that slot's default values. When the class-unit has one or more subclasses, the names of the subclasses are contained in the sub-menu of the menu-item *subclass*. This feature is quite useful since it shows the KEE ordering in which the subclasses were defined. Similar to a class-unit right-click, a member-unit right-click displays that member-unit's slots for which it has a value. The sub-menu's menu-items are that slot's values.

10.4 UNDERSTANDING THE ALGORITHMS FOR BUILDING TEMPORARY UNITS

The creation of Temporary Units cleanly falls into two types.
Type 1.
Methods that create units based upon slot value (Rules, Value Class, Metaconstraints)

Type 2.
Methods that create units based upon class membership (Sibling Class, Subclass)

10.4.1 Format of a KEE Rule

The LHS of a KEE rule is composed of a set of literals. The Rule Refiner is concerned with literals dealing with class membership, slot values, and lisp comparisons, also known class literals, slot literals and lisp literals respectively.

Examples of LHS literals that the Rule Refiner uses:

(?a is in pilot) -- *class literal 1*
(the sex of ?a is male) -- *slot literal 1*
(the age of ?a is ?b) -- *slot literal 2*
(lisp (> ?b 30)) -- *lisp literal 1*

Notice that while the slot value for age is not specified in *slot literal 2*, a boundary for the value can be found in the lisp literal by matching the *?b* variable in *lisp literal 1*.

10.4.2 Rule Refiner General Format

Both Type 1 and Type 2 algorithms for generation of Temporary Units follow a generalized format based on the class literals and slot literals used in the LHS of the rule. In Example 1 below, the class membership literal is *plane*, and the slot value literal is *speed*.

```
(if (?x is in plane)
    (the speed of ?x is ?y)
    (lisp (< ?y 741))
    then
    Some RHS action)
```

Example 1

10.4.3 Using a Type 1 Algorithm

Refining Example 1, using algorithm Type 1, creates TUs that are members of the class plane, each with a different slot value for speed. If the LHS contains more than one slot literal, then the rule refiner takes the cross product of all slot literal values used in the LHS (see section 10.4.5).

10.4.4 Using Type 2 Algorithm

Refining Example 1 using algorithm Type 2 creates TUs that are members of classes related to plane (either sibling class or subclass) with a slot value for speed of 741 (only if the slot *speed* was defined or inherited at that class level).

10.4.5 Taking the Cross Product of slot values

KEE Rule with two slot literals:

```
(if  (?x is in plane)
     (the speed of ?x is ?y)
     (the aileron_position of ?x is ?z)
     then
     Some RHS action)
```

Example 2

Using the Type 1 algorithm of Temporary Unit generation, the slot values gleaned from the KB for speed and aileron_position are [0, 110, 220, 440], and [up, down, stuck] (These slot values could have come from sources explained in 9.5). The cross product of these slot values are:

[stuck, 440]
[down, 440]
[up, 440]
[stuck, 220]
[down, 220]
[up, 220]
[stuck, 110]
[down, 110]
[up, 110]
[stuck, 0]
[down, 0]
[up, 0]

10.4.6 Naming the Temporary Unit

A Temporary Unit name is derived from its designated *member of* class with a unique number concatenated at the end of the class name. With Example 1, a Type 1 algorithm generates several Temporary Units. Each TUs is a member of the class *plane*, and given names like *plane1*, *plane2*,...,*planeN*.

10.5 IMPLEMENTATION OF TEMPORARY UNITS ALGORITHMS

10.5.1 KEE Value Class

By using the Value Class Concept an expert is able to partially or fully specify what values a slot may or many not have. KEE uses a restriction mechanism called the *Value Class*, which restricts permissible slot values to belonging to a specified class, range or set.

10.5.1.1 Example of a KEE Value Class

Unit: **PILOT**
in knowledge base **FLIGHT**

Created by combs on 11-14-89 23:56:07
Modified by cassell on 2-13-90 14:37:58
Superclasses: **FLT_PERSONNEL**

Member Of: **CLASSES** in **GENERICUNITS**
Members: **BOB, TIM, JACK, JOHN, SARA, TED**

Member slot: **AGE** from **FLT_PERSONNEL**
Inheritance: **OVERRIDE.VALUES**
ValueClass: # [Interval: [18 55]]
Values: UNKNOWN

This Value Class specification constrains the value for the slot age of a flt_personnel to 18 to 55.

10.5.1.2 Using Value Class Specification

Using information supplied by a Value Class specification, the Rule Refiner generates slot values for any slot literals used in the rule. This is a Type 1 algorithm for Temporary Unit generation.

Rule with slot literal age

```
(if  (?x is in flt_personnel)
     (the age of ?x is ?a)
     (lisp (<  ?a 54))
     then
     Some RHS action))
```

Example 3

Example 3 contains a slot literal for age. Using the Value Class specification from Section 10.5.1.1, the Rule Refiner generates two Temporary Units with slot:slotvalues of age:18 and age:55.

flt_personnel1:[age:18]
flt_personnel2:[age:55]

(The format used to describe a Temporary Unit is name:slot-list, where *name* is the unique name generated for that particular TU, and *slot-list* is a list of slot:slotvalues, in which *slotvalues* may be a list itself in the case of multiple values.)

10.5.1.3 Using Value Class to Refine a Rule

Suppose the RHS of Example 3 assigned flt_personnel to active duty. Now suppose there was

an active duty age range between 22 and 55. When the Rule Refiner displays the unit graph with

flt_personnel1:[age:18] (Highlighted)
flt_personnel2:[age:55]

the expert notices that the rule contains no lower bound for age. A corrected LHS might look like:

```
(if (?x is in flt_personnel)
    (the age of ?x is ?a)
    (lisp (< = ?a 54))
    (lisp (> = ?a 22))
    then
    Some RHS action))
```

Example 4.

10.5.2 DEVA Semantic (Meta) Constraints

The Meta Constraints algorithm, a Type 1 algorithm, uses the DEVA Semantic constraints *value_set* and *value_range*. Both value_set and value_range are used to extend the KEE Value Class Concept to cover specific situations. The Meta Constraint format, similar to the KEE rule format, uses the LHS to set up the situation, and the RHS to specify the constraint.

Example of a DEVA Meta Constraint:

```
if (?fj is in fighter_jet)
    (the pilot of ?fj is ?p)
    (?p is in flt_personnel)
    (the sex of ?p is male)
    (the age of ?p is ?a)
    then
    (value_range ?a (33 55))
```

Example 5.

This Meta Constraint specifies the age of a fighter_jet's male pilot to be between 33 and 55 inclusive.

Example 3, contains a slot literal for age. Using the Meta Constraint in Example 5, the Rule Refiner generates two Temporary Units with slot:slotvalues of age:33, and age:55

flt_personnel3:[age:33]
flt_personnel4:[age:55]

10.5.3 Using DEVA Meta Constraints to Refine a Rule

Suppose that the RHS of Example 3 asserted which health plan a flt_personnel would qualify for. Suppose that the legal values for a flt_personnel's age are as defined in Example 5. When the Rule Refiner displays the unit graph with

flt_personnel3:[age:33] (Highlighted)
flt_personnel4:[age:55]

the expert notices that though the rule is correct, the Meta constraint for flt_personnel is incorrectly specified because flt_personnel3's age, while within the Value Class range defined in Section 10.5.1.1, should be a boundary condition. A corrected Meta Constraint might look like

```
if (?fj is in fighter_jet)
   (the pilot of ?fj is ?p)
   (?p is in flt_personnel)
   (the sex of ?p is male)
   (the age of ?p is ?a)
   then
   (value_range ?a (22 55)))
```

Example 6

This demonstrates how the rule refiner can assist an expert in refining the Value Class, and Meta Constraints of a KB, not just the rule base.

10.5.4 Using the KB's Rule Base to Refiner Rules

The rule base is consulted for slot literal constants used in the LHS of any rule in the rule base. Any new slot literal constants that do not already exist in the fact base are used as slot:slotvalues when Temporary Units are generated. The slot values are taken without consideration of class membership in the rule from which they were derived.

Example of Rules with slot literals:

If the rule base only contained these two rules:

```
(if (?p is in f_111)
   (the mission of ?p is training)
   (the status of ?p is under-attack)
   (the age of ?p is ?a)
   (lisp (< ?a 10))
   then
   some RHS action)
```

Example 7

```
(if (?p is in plane)
   (the status of ?p is ?y)
   (the mission of ?p is classified)
   then
   some RHS action)
```

Example 8

Let Example 8 be the rule under refinement. Temporary Units that are created for Example 8 are all members of the class *plane* with slotvalues gleaned from the rule base. In Examples 7 and 8 above, the slot:slotvalues pairs are mission:[training, classified], and status:[under-attack]. Therefore taking the cross product of the slot:slotvalue pairs, two Temporary Units are created:

plane1:[mission:training, status:under-attack]
plane2:[mission:classified, status:under-attack]

10.5.5 Using Slot Literal constants to Refine a Rule

Suppose that when the expert created the rule used in Example 8 that the chance that a plane may be on a classified mission and come under attack was not anticipated.

plane1:[mission:training, status:under-attack] (Highlighted)
plane2:[mission:classified, status:under-attack] (Highlighted)

When the unit graph is drawn, the expert notices that plane2 should not have been highlighted, and that the rule under refinement needs to be modified.

10.5.6 Class—Subclass links

Both Sibling Class and Subclass Type 2 algorithms build Temporary Units based on the class literals used on the LHS of the rule.

10.5.6.1 Sibling Class

Similar to unit creation in the omission checker, Temporary Units are created for each sibling class of a class literal used in the LHS of a rule. The creation of Sibling Class Temporary Units assists the developer in determining when the class literals of a rule are too specific.

Using Example 3, let the sibling classes of *flt_personnel* be *ground_crew* and *atc_personnel*. The Rule Refiner creates one Temporary Unit for ground_crew, atc_personnel, and flt_personnel.

ground_crew1:[age:54]
flt_personnel5:[age:54]
atc_personnel1:[age:54]

Notice that each Temporary Unit created for Sibling Class contains a slot:slotvalue of age:54. Any slot literals used in the LHS of a rule that contain a constant are used in the creation of Temporary Units. Also notice that a Temporary Unit was created for the class flt_personnel. In construction sibling class Temporary Units, a class is considered a sibling of itself. This also assures that at least *one* Temporary Unit will be highlighted when the unit graph is drawn.

10.5.6.2 Using Sibling Class to Refine a Rule

Suppose that the RHS of Example 3 kept track of Social Security benefits for flt_personnel. The Social Security benefits for ground_crew and atc_personnel are the same as for flt_personnel (and thus do not each need unique rules). When the Rule Refiner displays a unit graph with

ground_crew1:[age:54]
flt_personnel5:[age:54] (Highlighted)
atc_personnel1:[age:54]

the developer notices that only flt_personnel5 is highlighted. This is an example of a rule that is too specific. Since *person* is the superclass of ground_crew, flt_personnel, and atc_personnel, the rewritten rule might look like:

```
(if (?x is in person)
    (the age of ?x is ?a)
    (lisp (< ?a 54))
    then
    (Some RHS action))
```

Example 9

10.5.6.3 Subclass

Creating subclass Temporary Units assists an expert in determining when the class literals used in a rule are too general. A Temporary Unit is created for each subclass of each class literal used in the rule under refinement. Any slot values of a Temporary Unit have come from the slot literal constants used in the LHS of the rule.

Using Example 3, let the subclasses of *flt_personnel* be *pilot*, *flight_engineer*, and *flight_crew*. The Rule Refiner generates Temporary Units for each class literal used in the rule. The TUs generated are:

pilot1:[age:54]
flight_engineer1:[age:54]
flight_crew1:[age:54]

Using Subclass to Refine a Rule

Suppose that the LHS of Example 3 was to remind flight_personnel to recertify once each year until age 54. Suppose that flight_engineers and flight_crews need not recertify and that only pilots are required to. When the Rule Refiner displays the unit graph with these three TUs added

pilot1:[age:54] (Highlighted)
flight_engineer1:[age:54] (Highlighted)
flight_crew1:[age:54] (Highlighted)

the developer notices that only pilot1 should be highlighted. This is an example of a rule that is too general. The rewritten rule might look like

```
(if (?x is in pilot)
    (the age of ?x is ?a)
    (lisp (< ?a 54))
    then
    Some RHS action))
```

Example 10

10.6 CONCLUSION

The process of refining a rule base is highly subjective, based on an expert's experience, intuition and practical knowledge. The DEVA Rule Refiner allows an expert flexibility to propose rules then, in a variety of ways, test those rules for excess generality or specificity.

CHAPTER 11

CONTROL CHECKERS

11.1 INTRODUCTION

The term *control checker* could have two possible interpretations with respect to rule-based systems. In the first interpretation, where we are concerned with entire rules, the developer could specify the interactions that should be preserved between the rules in the KB in the form of control constraints, and the checker could alert him of possible violations. In the second interpretation, the control checker would verify that the KBS always operated within the control constraints. The first interpretation is the one adopted by DEVA.

The important thing to remember when examining the various control checkers is that they flag possible violations of various control constraints. Because of decidability issues, these checkers do not check for complete compliance.

Any Turing machine can be reduced to a set of facts, a set of rules and its nonmonotonic inference mechanism, in other words a knowledge-based system (the nonmonotonicity is not strictly necessary, it is simply easier to envision where tape rewrites are concerned). The various rule firings correspond to state transitions, and the tape input is the body of facts. Assume we have an algorithm that checks for complete compliance with control constraints, and that we have asserted the control constraint *necessary(initial-situation,final-situation)*. This constraint means that the final situation will necessarily follow from the initial situation. Thus, by our reduction, we also have an algorithm that solves the halting problem for Turing machines. In the general case, a compliance-oriented control checker is an undecidable problem. The DEVA project is not tasked with solving the halting problem; however, this does not preclude us from indicating interesting, significant violations of the control constraints, either independent of or dependent upon the current input (facts).

The control checker has several components. These are sequence, exclusion, necessity, conditional, rule interference and enhanced cycle checking. Each of these components is in some way concerned with whether or not rules in the KB can be satisfied under a given set of conditions. These conditions may range from a collection of facts which are true, the previous firing of other rules, to various combinations of facts and rule firings.

11.2 ASSUMPTIONS

It is useful for us to make explicit our working assumptions for the control checker. This is equivalent to stating the *axioms* upon which we will base our work. These assumptions are:

Independence. The various control checks are performed independent of any given conflict resolution strategy. In other words, we do not presume to know the specific order in which rules will be examined during inferencing.

Noncompliance. The control checker will only be able to indicate with complete certainty when a given control constraint is violated. The control checkers will not be able to say with absolute certainty that the present KB will always be in compliance with a given control constraint.

Nonprocedural. Procedural calls, appearing in the LHS or RHS of a rules, are ignored at

this time. Only the declarative aspects of KB rules are addressed.

Revision. The RHS actions which the control checker will concern itself with are *add* and *delete*, and a *modify* action will be treated as a *delete* followed by an *add*.

Reducibility. Since a KB consists of atomic formulas (facts) and conditional formulas (rules), and the previous assumption excludes functional formulas, any control specification must eventually be expressed in terms of atomic or conditional formulas. Any abstract property of the KB which is used in a control constraint can be replaced by the formulas which satisfy that property, thus the control checker will concern itself only with those formulas (facts and rules).

11.3 RULE INTERFERENCE

The interference checker determines violations of an implicit control constraint. Whenever a developer produces a rule, he expects that rule to be used under some set of circumstances (fact-scenario) that he believes is likely to occur. If it is impossible to use the rule because of its interactions with other rules in the KB, then the developer's implicit constraint of *rule usage* is violated.

Essentially interference is the inverse of rule-inconsistency detection. In rule-inconsistency detection, we wish to detect the derivation of conflicting values from the rules and a likely fact-base scenario. In interference detection, we want to show cases where the only way to prove a valid goal is via the rules and an inconsistent (i.e., unlikely) fact-base scenario. Consequently interference can result in unexpected proof-failures and useless rules. Therefore DEVA will act as a diagnosis tool to explain why the rule "did not work". Rules with interference problems can be satisfiable when the expert system uses an unsound, procedural form of inference. This is the case with many production rule systems. For these systems, our interference checkers will point out to the developer where he is relying on unsound methods to do inference (i.e., the declarative reading of the rules has been compromised). Our check for interference is performed using only the static facts and rules of the knowledge base. We have also devised a strategy to model the operational semantics of nonmonotonic rule-bases.

11.3.1 Proof-Residues And Ramifications

To generate the assumptions, we use residue resolution. A proof residue of a goal is a set of dynamic conditions missing from the fact-base which, if present, would entail the goal. To determine a proof residue for a goal, we generate a consistent proof-tree of the goal using backward chaining. A proof-tree is consistent provided conflicts do not exist between the literals in the tree. For example, a conflict exists in a tree which assumes *A* and *not(A)* or ?X > 10 and ?X < 5. The special case of determining the relationships between numerical comparisons is discussed in more detail in section 11.5 of this chapter.

Backward chaining is bounded by a user-specified proof-depth to ensure termination and to provide an upperbound on the effort to expend. The leaves of a proof-tree constitute a proof residue. Since there may be multiple proof-trees, there can be several proof-residues for a goal. All literals derivable from a proof residue are the ramifications of that proof residue. The original goal is a *necessary ramification* of the proof residue, as are all the intermediate literals in the proof-tree used to derive the goal from the proof residue. There can also exist *extraneous ramifications* which are additional literals derivable via the proof residue, but which

are unnecessary to prove the goal. Consider the following rules:

C → not(E) A → B B → C D → E E → F C and F → G

The proof-residue of the goal C is A . The necessary ramifications of A are [B,C] , while its extraneous ramification is not(E) . The proof-residue of the goal F is D, and the necessary ramifications of D are [E,F].

Given two subgoals *g1* and *g2* with proof-residues *Rg1* and *Rg2* respectively generated from consistent proof-trees, the subgoals *interfere* if *Rg1* and its ramifications conflict with *Rg2* and its necessary ramifications. Therefore an interference exists in the conjunctive goal (C and F), since the extraneous ramification not(E) produced from proving C conflicts with a necessary ramification of the proof-residue of F. If an extraneous ramification of one subgoal's residue conflicts with an extraneous ramification of the other's residue, then we will have created an instance of rule-inconsistency. Since the conflict does not defeat a necessary literal in one subgoal's proof-tree, it is not interference (but it is still a problem).

A subgoal interferes with itself if there does not exist a consistent proof-tree for the subgoal. Consequently, any rule containing such a subgoal would be unsatisfiable.

11.3.2 Weak And Strong Interference

For rules containing several antecedents, we provide two types of interference checks. If we can show the existence of two subgoals in the preconditions of a given rule such that all consistent proofs of one of the subgoals interfere with any consistent proof of the other, then we have demonstrated an instance of *strong interference.*

There may be times when we are interested in finding any form of interference that might exist between any two literals appearing in a given rule's LHS. This would be referred to as a check for *weak interference.* Each case of weak interference contributes to a possible case of strong interference. When strong interference is detected, there definitely is a problem. By showing cases of weak interference, we indicate how strong interference may occur.

11.3.3 Using Strong Interference To Validate Constraints

Above we stated that strong interfence creates a problem. The designer can use this condition to do simple KB validation. DEVA allows the developer to specify ad-hoc constraints on KB behavior. For example, the designer is allowed to specify the rule

(alpha and beta) → incompatible

to indicate that the *alpha* and *beta* conditions should never be true at the same time. If it could be proved that a case of strong interference exists for a constraint of this form, then we will have validated the KB with respect to the constraint (assuming a sound inference technique).

11.3.4 Handling Nonmonotonicity During Interference Detection

When a rule-base allows deletions on the RHS of rules, one needs to model the operational semantics of the rules to accurately evaluate the rule-base for anomalies. In Chapter 12 on nonmonotonicity, interference detection is used as a detailed example of how one would

model the operational semantics of rules during anomaly detection.

11.3.5 Consistent Numerical Constraints

In the residue analysis approach to verification, one generates the conditions necessary to infer an anomaly and then tries to determine whether these conditions are likely to occur simultaneously; therefore, one must be careful not to require an inconsistent residue to demonstrate an anomaly. This would be useless because from an inconsistent set of beliefs, one in theory can prove anything. In the simplest case, it is a matter verifying that one never assumes preconditions such as *alpha* and *not(alpha)*. A more subtle check is one that verifies that all numerical comparisons are consistent. As a trivial example, one would not want to assume ?Z > 5 and ?Z < 3.

An efficient numerical-comparison consistency module has been built, which can be used by any residue-based checker. It is also used by the enhanced cycle-checker to screen out phony cycles. By efficient, we mean it should scale up nicely; however, on trivial problems a naive approach may be faster (less overhead from algorithmic simplicity).

The problem of determining the consistency of constraints of the form *?X R ?Y* (where R is from [< , > , ≥ , ≤ , = , $\neq$]) is much more interesting and difficult than it appears on the surface. It requires an $O(N^3)$ algorithm (N the number of variables) to accomplish the task, since transitive closures must be computed to do complete checking. For example, the set

[Z > = 2, Z = < 2, Z < X, X < W, X< W2, W2 < W3, W < 10, W3 < 1]

is inconsistent. The first two conditions force Z= 2 and consequently the condition Z < X becomes 2 < X. Via transitivity from 2 < X, we can infer 2 < W3. But this condition is incompatible with the existing condition W3 < 1. This examples points out some of the difficulties in performing this type of check (i.e., handling transitivity and the inferred binding of a variable from existing constraints). In addition, if cycles exist from the transitivity relationship, the facts are inconsistent (i.e., X < Y and Y < X is inconsistent) unless the cycle involves the = < relation (i.e., X = < Y and Y = < X forces the condition X = Y).

11.3.6 Interference Detection Methods

For illustrative purposes, it is useful to examine the monotonic case of backward-chaining interference (BCI) detection first. The method for detecting interference in the monotonic case is relatively straight-forward but it is an integral part of generating a consistent residue for any nonmonotonic method. Related techniques have been proposed that make use of semantic information to optimize user-level queries in deductive databases by, among other things, determining when such queries are unsatisfiable. Backward-chaining interference detection in effect treats the LHS of each rule in the knowledge base as such a query. The emphasis here is not so much on user-level query optimization, but instead on highlighting possible anomalies for each respective rule.

The method begins by examining the LHS of each rule in the KB as a conjunctive goal. This is referred to as the original LHS, to differentiate it from the LHS's of other rules involved in the process of proving the original LHS.

The residue of each antecedent is generated by backward inference from the antecedent to the base set of assumptions necessary for it to be true. The construction of the proof residue

is interleaved with the check for interference. Whenever the entire LHS of a rule is satisfied during this inference, the proof tree is updated with the current RHS goal occurring in that rule. Then this goal's residue and necessary ramifications are examined to see if they contradict the residues and necessary ramifications of any of the other original LHS goals (the types of possible conditions are outlined in the section dealing with residues and ramifications). If such a contradiction is found, it is reported to the user as a case of weak interference. No further exploration of that particular proof residue is then necessary. Otherwise, the inference proceeds to the maximum allowed depth, trying to satisfy the rest of the goals. Detecting strong interference under this scheme is a matter of recording the presence or absence of interference in each alternative proof residue attempted. If every such proof residue attempted has weak interference, then the developer is warned of the presence of strong interference for the respective rule.

It is sufficient to consider only the original LHS when looking for contradictions in each proof residue. We can ignore possible interference in the other LHS goals involved in the proof residue. The interference check iterates through all of the rules in the knowledge base so these other LHS goals will eventually be examined as the original LHS and any interference will be detected at that time.

Backward-chaining interference detection can be augmented in the presence of nonmonotonic revision with only a small overhead by incorporating a state representation that captures the effects of local extraneous ramifications. The rule A,B -> D,E,delete(F) appearing in a knowledge base can be equivalently expressed as three separate rules:

A,B -> D A,B -> E A,B -> delete(F)

Thus when proving D, its local, extraneous ramifications are E and the deletion of F. If a separate rule A -> G appears in the knowledge base, then G is considered a nonlocal extraneous ramification of D.

This method relies upon a fixed evaluation strategy and concept of state. For most knowledge base systems that offer backward-chaining, the state *S* is cumulative according to goal ordering. Whenever a goal is proven, its local ramifications are used to update the state. For backward chaining in expert systems, conjunction between antecedents is usually implemented by sequentially examining the antecedents. Once an antecedent is satisfied, the next is examined in the same manner. Even though satisfying the next antecedent may cause the previously examined antecedent to become unsatisfiable because of nonmonotonic properties, the previously satisfied literals are not re-examined to see if they are still true in the final state *S*. This is fine for a procedural interpretation, but when the antecedents of a rule are stated conjunctively in a declarative context, they must all be satisfiable in the same state of reasoning.

Essentially interference detection is achieved by cumulatively updating the state and re-examining previously satisfied literals to see if they still hold true. This form of detection indicates when rules are unsatisfiable, and highlights the case where the LHS of a rule is procedurally satisfiable, but its declarative context is not upheld.

The antecedents of each rule in the knowledge base are examined in the same manner as in the monotonic case of backward-chaining interference detection. The backward inference proceeds in an attempt to satisfy each of the goals in the LHS in left-to-right order. Whenever the antecedents of a rule necessary for the proof are satisfied during this inference,

the state representation, in the form of a proof residue, is updated with the goal proven and any local ramifications (or side-effects, in the case of Prolog). This is a limited form of forward chaining local to the rules necessary for the proof. At this point, the state representation is examined to see if it contradicts any of the original LHS goals. If such a contradiction is found, it is reported to the user as a case of weak interference.

The advantages gained from interleaving the construction of a proof residue with state update and interference check is that no separate pass over the proof residue is required to detect contradictions. Other advantages are those inherited from normal backward inference: only those rules which participate in the proof of the original LHS are examined, increasing efficiency. Since all these rules directly take part in proofs, coherent explanation traces of how weak interference occurs can be easily collected for presentation to the developer.

A disadvantage of this augmented backward-chaining detection is that it only takes into account local extraneous ramifications appearing in the RHS of rules actually involved in the proof-residue. This is entirely appropriate for backward-chaining inference systems, but there are other ramifications possible under forward inference. These extraneous ramifications can lead to contradictions as well, and under systems which have coherent forward inference, this check may not detect all cases of interference. However, determining all the ramifications of a given residue is an intractable problem in the general case. Some upper bound on the effort to expend must be set. The tradeoff is completeness versus computing time. In this case the upper bound is limited to local, extraneous ramifications so that a reasonably quick check for interference is available for the developer, even if he is relying on a forward inference system.

11.4 EXPLICIT CONTROL CONSTRAINTS

Each control constraint requires a separate checker within the general framework of the control checker. The *precedes* constraint indicates the sequence in which rules should fire, and is associated with the Sequence checker. The *excludes* constraint indicates that certain rules should exclude other rules, and is associated with the Exclusion checker. The *necessary* constraint indicates that using certain rules will necessarily lead to the use of another set of rules, and is associated with the Necessity checker. Finally, the *exception* constraint indicates that a certain body of rules will always be used except when a given condition is true. This constraint is associated with the Conditional checker.

The sequence, necessity, exclusion, and conditional checkers all rely on the same control constraint collector to preprocess their respective control constraints. This collector takes a control constraint that deals with the abstract properties of rules in the KB and maps the abstract properties to the actual sets of KB rules that satisfy those properties. It uses the property arguments to collect the sets of rules they represent. The abstract control constraints are thus reduced to more concrete control constraints involving sets of rules which actually exist in the KB. Each of these checkers then uses this reduced form of control constraint to uncover any violations.

To support exploration of the KB using the control checkers, DEVA provides a Control Constraint Selector menu option. This selector provides menus of the four constraint types, and menus of the many property types. Using these menus and a text-entry area for each argument, the developer can easily build up a control constraint. Once this is accomplished, the developer can list, assert or retract this control constraint. To aid the text-entry of arguments, the developer can also view examples, once the constraint and property types are

selected.

11.4.1 Properties Of Rules

To represent the arguments for the control constraints, DEVA provides meta-predicates that represent the properties of rules. Each property denotes a set of rules that the respective control checkers use when checking a specified control constraint.

Since the control constraint collector takes property arguments and replaces them with sets of rules, it is necessary to further explain what the properties mean in terms of ordinary KB representation. After each type of property, the corresponding low-level KB constructs involved in collecting the proper rule sets are explained. There are two classes of properties; *unbound* and *bound.*

In the following *unbound* property specifications, rules which match the properties are simply collected - this does not mean the rules are actually bound to the property specifications. During the actual control checking that follows, the rules may not have the same actual bindings as the properties. By looking for a match, we have found rules which can, but do not necessarily have now, the same bindings as the properties. Essentially, rules which are at least as general as the properties themselves, or more general, are collected.

Name: This is probably the simplest, and most restrictive property. In this case, only those KB rules which have excactly this name are collected as part of the rule set.

lhs(LHS): The lhs/1 property collects any rules which have a left-hand-side that matches at least those literals in its LHS argument.

rhs(RHS): The rhs/1 property collects any rules which have a right-hand-side that matches at least those literals in its RHS argument.

rule(Name,LHS,RHS): This is the most general property available. Any rule that satisfies this "template" is part of the set of rules collected. Rules collected must match at least those literals appearing in the LHS and RHS arguments, and if a name is specified, the name must be exactly matched.

rule_class(Class): Within KEE, rules themselves belong to classes. This property naturally groups the rules into sets, and so control constraints using rule class properties will be useful.

depend(LHS): The depend/1 property is used to collect any rules which depend on any of the literals in the LHS argument. In this case, any rule which has at least one, but not necessarily all of the literals in LHS, is collected.

activity(RHS): The activity/1 property collects rules which carry out a given activity. In this case, any rule which has at least one, but not necessary all of the literals contained in the RHS argument will be selected for membership in the rule set.

transition(LHS,NewLHS): Rules in the KB can be collected by determining the state transitions that they achieve. A state transition can be expressed by a group of LHS literals true in the previous state, and LHS literals that are true in the following state. The property used to represent this is transition/2, in which the first argument is the initial

state and the second is the resulting state. Rules whose LHS literals are a subset of the *previous* literals and whose RHSs directly satisfy all the *following* LHS literals are collected as elements of the rule set.

slot_value(Slot,Value): The slot_value property selects rules based on the values they assign to slots. The slot value computations are only performed on the RHS of rules. Any rule which has a RHS literal that manipulates the slot value is an element of the rule set.

slot_request(Slot,Value): Rules which on their LHS make a specific request for a slot value can be collected into a set using the slot_request property.

predicate(Lit): Collection of rules based on the predicates they define is accomplished by using the predicate/1 property. The predicates defined by a rule would necessarily be one of the rule's RHS actions. Any rule containing the RHS actions becomes an element of the set being collected.

purpose(Meaning): The property purpose/1 denotes the purpose of rules. This could be quite abstract. DEVA will allow the developer to attach purpose descriptions to the KB rules (or use the native KB system's *purpose* field). Normal matching of these purpose descriptions will then be used to collect sets of rules.

priority(Level): The priority of a set of rules is an integer associated with all rules in a given rule set. All rules with a given priority Level are collected.

The following *bound* properties can be used not only to select the rules for a control check, but to also give those rules a specific binding during the following control check. The control check then becomes more specific to the rule properties that the developer states. The violations uncovered will not be as general as those found using non-binding properties, but they may reveal more to the developer about how the violation arises.

bound_rule(Name,LHS,RHS): The bound_rule/3 property is much the same as the rule/3 template, except that it also binds with each rule found in the set during the course of any of the control checks.

bound_lhs(LHS): The bound_lhs/1 property expects a list of LHS literals just as the lhs/1 property does, but also binds with the rules during control checking.

bound_rhs(RHS): The bound_rhs/2 property expects a list of RHS actions just as the rhs/1 property does, but also binds with the rules during control checking.

situation(LHS): Situations when the rules can be applied are denoted by situation. Each situation is denoted by LHS literals. A situation is a subset of the KB state, and many different KB states could satisfy a given situation. Any rule whose LHS is contained in the situation would be considered as part of that set.

operation(RHS): Operations defined by the rules are represented by operation. Operations are really RHS literals. Individual rules containing any of these *operations* are elements of the rule set.

11.4.2 Property Filter

A rule-property filter that deals with the property arguments of the control constraints indicates when the developer has supplied it with nonsensical control constraints. For the *exclusion* and *precedes* constraints, the sets of rules defined by abstract properties must be non-intersecting in order for them to make any sense. Another example of this is when a control constraint is defined such that a situation (facts true in a KB state) must precede itself. Also, this filter can indicate when no further checking is necessary, because the properties are stated so as to be trivially true. Consider the case where the *necessity* control constraint states that rules (in second set) which are satisfied in the situation where literal A is true must necessarily be satisfied by rules (in the first set) which are fired in the situation where both A and B are true. In this case, as long as there exist rules satisfying both these situation properties, the *necessity* check is unnecessary because rules in the second set are trivially going to be satisfied whenever rules in the first set are satisfied.

11.4.3 Residue Proofs Applied To Explicit Control Checks

The sequence, necessity, exclusion, and conditional control checkers rely heavily on the proof-residue methods developed for the interference checker. Most of the techniques employed by these checkers are explained in the section on rule interference (11.3). The following sections relate these techniques to the specific checkers.

A residue is that body of assumptions which is necessary to prove a given RHS goal. One can think of a residue as filling in the missing facts in a partial or incomplete KB. During the course of KB development, it will often be the case that the developer wishes to determine if his body of rules can lead to anomalies, and yet has not built a complete set of facts that exercises the rules in the proper form to detect errors because the body of rules itself is in flux. During the development of a KB, its rules and the facts these rely on are subject to revision.

The control checkers actually keep track of more than just assumed facts during the course of inference. Whenever the LHS of a rule is satisfied, all of its RHS actions are used to update the residue. These are known as ramifications. The residue and ramifications together form a state which is sometimes also referred to as a residue. In some cases the primitive, assumed facts are referred to as findings, and the ramifications that result as hypothesis, since they rest on the primitive facts. These terms should be familiar to those acquainted with truth-maintenance systems. During the course of inference the residue is constructed by assuming literals that occur only as findings (no hypothesis literal is directly assumed true or false). That is, literals appearing on the RHS of a rule must be proven using the rule and are not directly assumed true. This is a reasonable simplification, but it can be erroneous in cases where a hypothesis literal is sometimes expected to appear as a free-standing fact.

Different variations upon residue proofs are possible. A useful variation is one that allows the residue to be augmented with a collection of literals. The developer may require that a reduced body of static facts should be used during inference. In this case the residue is initially augmented with these static facts, and no assumptions contrary to them are allowed. Static facts are usually those which are inherently part of the domain description, such as facts describing connections in a circuit domain; they are essentially treated as rules with a trivial, null precondition. On the other hand, dynamic facts (e.g. specific slot values) contained in the knowledge base at any given time should be ignored when using residues. An example of dynamic facts would be input values for a circuit component.

11.4.4 Sequence Checker

The sequence checker determines violations caused by rules firing out of sequence with respect to the *precede(R1,R2)* control constraint. Since the control checkers do not assume any particular conflict resolution strategy, such a violation is flagged when it is possible for rules in the second set R2 to fire independently of rules in the first set R1.

For the sequence constraint to be upheld, there must be at least one dependency between each rule in the second set and the rule(s) in the first set. This dependency may be one or more literals participating in an inference chain which do not appear as facts in the current fact base, or on the RHS of any rule which can be satisfied independently of rules in the first set. To determine if dependency exists, the sequence checker first uses the connection graph as a filter for violations. If there exists a rule in the second set that is not connected to any rule in the first set, there can be no dependency that will enforce sequence, and the second rule is shown as a violation.

The connection between rules in the second set and rules in the first set is a necessary, but not sufficient condition for showing such a dependency. There should not exist any other way of satisfying a rule in the second set independent of the rules in the first set. To further determine violations, a residue proof is attempted for each rule remaining in the second set after the first filter. Rules contained in R1 are excluded during this proof process. If such a proof is successfully constructed (to a given maximum depth limit) then the developer is warned of a possible control violation, by displaying the first residue that allows firing a rules in R2 independent of any rule in R1.

11.4.5 Necessity Checker

Each *necessary(R1,R2)* constraint indicates that whenever a rule in the first rule set R1 fires, any rule in the second rule set R2 should be able to fire. The necessity checker determines violations caused by rules in R2 which cannot fire, in contradiction to the constraint. In other words, if any rule in R1, defined as necessarily triggering (directly or indirectly) all rules in R2 actually causes them not to fire or is not sufficient to fire rules in R2, then a violation occurs.

Since any consistent residue for any rule in R1 should be sufficient for any rule in R2 to fire, for each rule in R1, a consistent proof residue is generated. Then, using this proof residue as a starting point, a proof is attempted for each rule in R2. If a contradiction is required to prove a rule in R2, or if the necessary conditions for proving a rule in R2 are unsatisfiable because of actions carried out during the proof of R1, then this pair of rules, one in R1 and the other in R2, is reported as a violation, indicating that in this fact-scenario where the rule in R1 fires, the rule in R2 could not be satisfied. The proof residue of R1 for this specific violation then becomes available for the developer to browse.

11.4.6 Exclusion Checker

For the exclusion checker, violations are determined when rules in a second set R2 can possibly fire even though they are defined as excluded, using the *exclude(R1,R2)* constraint, by rules firing in a first set R1. For one set of rules to exclude another set of rules, anti-dependencies or contradictory requirements should exist between them. An anti-dependency is a RHS action which actively prohibits the satisfaction of a LHS condition. Examples of this are *add(A)* whenever *cant_find(A)*, negation-as-failure, appears in the LHS of rules in the

second set, or *delete(A)* whenever *A* appears as a condition in rules in the second set.

Contradictory requirements are cases where proving a rule indirectly requires the presence of a contradiction, as A and not(A), or X < 5 and X > 10, or X ∈ alien and X ∈ earthling, where *alien* and *earthling* are incompatible classes.

The residue proof procedure (as outlined in the interference section) is used to see if each rule in the first set excludes all rules in the second set. The initial residue is augmented with the LHS and RHS literals of a given rule (R1) in the first set. Then a residue proof (to a user-specified depth) is attempted for each rule R2 in the second set. If such a proof is possible, the violation is presented to the developer, indicating that rule R1 does not, with this residue, exclude the firing of R2.

11.4.7 Conditional Checker

The conditional check flags violations whenever the condition stated in the control constraint *exception(Condition,RuleSet)* is true, but it is still possible to fire a rule(s) in the specified set of rules. This constraint relies on anti-dependencies and contradictory requirements much like the exclude(R1,R2) constraint, and thus the method for determining violations in the conditional checker is similar. An initial residue is augmented with the literals appearing as the first argument to *exception/2* (the conditions). A residue proof is then attempted for each rule Rn in the set of rules collected by the control constraint collector, using the second rule-property argument of exception/2.

11.5 ENHANCED CYCLE CHECKER

We consider rigorous cycle checking a control check because cycles are one form of rule base system control, the other being the If-Then action.

The original cycle checker, which is part of the Structure Checker (see section 5.6 of this report), is an efficient first screen which finds a relatively small set of candidate cycles. However, the original cycle checker does not make use of all the available semantic information regarding legal values and data types for slots and class typing information. In addition, the original cycle checker does not check for the presence of inconsistent assumptions involving two literals in a cycle. That is, assuming both literal *P* and *not(P)* true may invalidate the cycle. Furthermore, if the cycle contains any inequality comparisons, the original cycle checker does not check that the set of comparisons is consistent. Clearly, a cycle can not be genuine if one literal in the cycle tests for X less than 5 and another literal in the same cycle tests for X greater than 10. Finally, the original cycle checker does not take into account the propagation of variable bindings.

To enhance the cycle checker and to take advantage of the available semantic information only requires a modification to the existing cycle checker. Once a set of candidate cycles is produced by the original cycle checker, this set is passed on to the enhanced cycle checker which acts as a second filter. Based on the propagation of variable bindings and the existence of any semantic constraint information, the enhanced cycle checker verifies that a direct or indirect candidate cycle is genuine. In our implementation, the enhanced cycle checker operates independent of the fact base. That is, no facts need to exist in the data base for the enhanced cycle checker to correctly verify the validity of possible cycles.

11.5.1 Cycle Verification

The detection of phony cycles is essentially a two-step process. The first step involves the propagation of variable bindings throughout the rules of a candidate cycle. At this point in the analysis, several of the phony cycles may already be eliminated before we even consider the existing semantic constraint information. The second step involves the use of the semantic constraint information to eliminate more phony cycles from the candidate set of cycles. It is at this stage of the cycle verification process where most of the phony cycles are eliminated based on illegal values and data types, unrelated classes, inconsistent assumptions, and inconsistent sets of inequalities.

11.5.1.1 Verification based on the propagation of variable bindings

Before information related to existing semantic constraints is utilized, a candidate cycle may be invalidated when variable bindings are propagated across the RHS to LHS connections between rules in a cycle. For example a predicate *p* on the RHS of a rule in a cycle may not unify with the predicate *p* on the LHS of the next rule in the cycle because of differences in the variable bindings. To illustrate how the propagation of variable bindings can invalidate a cycle, consider the following example of an indirect cycle [R1,R2,R3,R1]. (Note in this and all following variable binding examples: variable IDs, e.g., X and Y, are relevant only within the context of a rule; the propagation of variable bindings between rules is based on predicate ID, e.g., p and q, and argument position with respect to the predicate.)

R1: p(X,Y) → q(X,Y)

R2: q(Z,yes) → r(Z,no)

R3: r(W,no) → p(W,no)

The original cycle checker reports the cycle as genuine since the literals involved in the three RHS to LHS connections unify with each other. That is, *q(X,Y)* unifies with *q(Z,yes)* because we can bind the variable *Y* to the constant *yes*. The unification of the RHS literal *r(Z,no)* in rule R2 with the LHS literal *r(W,no)* in rule R3 is trivial. Finally, the literal *p(W,no)* on the RHS of rule R3 unifies with the literal *p(X,Y)* on the LHS of rule R1 since the constant *no* can be substituted for the variable *Y*. However, after we propagate the variable bindings across the rule connections, we get the following as one possible interpretation.

R1': p(X,no) → q(X,no)

R2': q(Z,yes) → r(Z,no)

R3': r(W,no) → p(W,no)

We can now see that the above indirect cycle is phony. Note that the value of the LHS literal *q* in R2' must be *yes* because of the rule definition. Yet, once we propagate the variable bindings, the value of the RHS literal *p* in R3' forces the value of the LHS literal *q* in R2' to be *no*. Thus, the cycle is phony, and the enhanced cycle checker will report the phony cycle at this stage of the analysis.

11.5.1.2 Verification based on semantic constraint information

Assuming a candidate cycle passes the initial stage of variable binding propagation, we next use existing semantic constraint information in an attempt to eliminate the cycle. Essentially, three groups of semantic checks are implemented which analyze the rule literals of a potential cycle for inconsistencies regarding legal values and data types, class membership, assumed literals, and inequalities.

The first semantic check analyzes the rule literals of a cycle searching for illegal values and incompatible data types. This semantic check utilizes facts in the data base of the form *semantic_constraint(< class> ,< slot> ,< restriction>)*. In this representation, *< class>* is the class of objects to which the semantic constraint applies, and *< slot>* corresponds to the predicate name associated with a literal. The third argument, *< restriction>* , can be either *interval(inc,x,inc,y)* meaning that the slot value is bound by [x,y], *legal_values([x1,...,xn])* meaning that the slot value must be in [x1,...,xn], *kee_datatype(Type)* meaning that the slot value must be the KEE data type *Type*, or *datatype(Type)* meaning that the slot value must be the DEVA data type *Type*. To illustrate the role that legal values play in the verification of a cycle, consider the following example of an indirect cycle [R4,R5,R4]. Assume that the data base contains the following semantic constraints imposed on the slots *p* and *q*.

Given:

semantic_constraint(classA,p,interval(inc,0,inc,10))
semantic_constraint(classA,q,interval(inc,6,inc,15))

R4: p(X,Y) → q(X,Y)

R5: q(W,Z) → p(W,Z)

From the semantic constraint information, we see that the value of slot *p* is bound by [0,10] while the value for slot *q* is bound by [6,15]. Therefore, the indirect cycle [R4,R5,R4] is genuine only when Y = Z is bound by [6,10]. As an example of how data type specifications can affect cycle verification, consider the following candidate direct cycle [R6,R6].

Given:

semantic_constraint(ClassA,p,datatype(integer))
semantic_constraint(ClassA,q,datatype(person))

R6: p(X,Z), q(Y,Z) → p(a,Z)

The semantic constraints specify that the value Z for slot *p* must be an integer. Yet, the value Z for slot *q* must be a member of the class *person*. Thus, we have a conflict in data types which invalidates the cycle.

Another semantic check is concerned with the presence of inconsistent assumptions in a cycle. For example, consider the following candidate cycle [R7,R8,R7].

R7: p(X,Y), r(W,Z) → q(X,Y)

R8: q(X,Y), not r(W,Z) → p(X,Y)

We notice that R7 contains the literal *r(W,Z)* on the LHS while R9 contains its negation,

namely, *not r(W,Z)*. In this case, the cycle is reported as possibly phony since the literals r(W,Z) and not r(W,Z) must both be satisfiable for the cycle to exist.

The third semantic check analyzes the rule literals of a cycle and searches for a set of inconsistent inequalities. For instance, consider the following case of a phony indirect cycle [R9,R10,R9].

R9: p(X,Y), less(Y,5) → q(X,Y)

R10: q(W,Z), greater(Z,10) → p(W,Z)

Since the variable Y in R11 unifies with the variable Z in R12, the cycle is reported as phony because the value of Y = Z can not be both less than 5 and greater than 10.

11.5.2 Implementation

The enhanced cycle checker acts as an add-on module to the original cycle checker. Hence, the first step in the cycle verification process is to obtain either a list of direct cycle candidates or a list containing a series of rule names that represent an indirect cycle candidates from the original cycle checker. To verify that a potential cycle is actually genuine, we need to first construct a list containing the LHS and RHS literals of each rule in the cycle. This list is constructed by traversing the cycle twice to make sure that all variables that can be bound are bound during the second traversal of the cycle. For a cycle [R1,R2,R1], the list of cycle literals takes the following form:

[LHS1-RHS1,LHS2-RHS2,LHS1-RHS1,LHS2-RHS2,LHS1-RHS1]

While the list of cycle literals is being constructed, any literals encased by the keyword *lisp* which equate a variable to a value are evaluated and removed from the resulting list. The new value is substituted for the variable throughout the LHS and RHS of the rule.

The next step is to propagate the variable bindings across the RHS to LHS rule connections. This is done using *iselem(Lit,NextLhs,Justification)*, where *Lit* is the LHS equivalent of the current rule's RHS literal which forms the RHS to LHS connection from the current rule to the next rule in the cycle. Using *iselem/3* rather than *member/2* allows us to correctly handle synonymous literals that may be present in the cycle. It should be noted that the point in the rule chain at which we start to propagate variable bindings contributes to the success or failure of the variable binding propagation part of the analysis. To ensure a unique report of either a genuine or phony cycle, variable bindings are propagated across the rule connections beginning with the first rule in the chain. The process continues until the bindings have been propagated through the rule chain twice. As we mentioned earlier, several of the candidate cycles may be eliminated at this point in the analysis before the semantic constraint information is even considered.

In the third step, we extract a set of inequality literals from the list of cycle literals, if any exist. An inequality literal has the following form:

lisp([Relation,Arg1,Arg2]),

where *Relation* is in *[less,lessthanorequal,greater,greaterthanorequal,equal,notequal]*. We attempt to evaluate both *Arg1* and *Arg2*. If one of the arguments of the relation can not be evaluated,

the relation is excluded from the resultant set of inequality literals. Otherwise, the nonvariable arguments are evaluated, and a new inequality is formed which is then appended to the final result. We then use the predicate *consistent_constraint_set/1* to check the consistency of the set of inequality literals.

After any existing inequality literals have been checked for consistency, we next need to skolemize the list of cycle literals to prevent unintended bindings when the semantic checks are performed. Skolemization is done using the *numbervars/3 predicate*, which binds unique constants to free variables. As an example of skolemization, consider the following direct cycle:

R13: p(X,Z), q(Y,Z) = > p(Y,Z)

After the literals *p* and *q* are skolemized, R13 becomes:

R13': p(sk1,sk2), q(sk3,sk2) = > p(sk3,sk2)

We now have meaningful constants that can be used to represent legal ranges of values.

The final step in the cycle verification process utilizes existing semantic constraint information to eliminate even more candidate cycles. These semantic constraints are supplied by the developer and impose restrictions on the rule literals. It is at this point in the cycle analysis where we check that each of the cycle literals contains a legal value and that each value is the proper data type. In addition, class typing information is used for semantic checks involving two or more *member_of* literals containing the same instance as the first argument. In such a case, we need to check that the classes (the second argument of each *member_of literal*) are related. Furthermore, if a potential cycle consists of rules containing both a literal and its negation, the cycle is reported as possibly phony unless of course the cycle fails a subsequent check. In the latter case, the cycle is reported as definitely phony.

As stated above, the enhanced cycle checker has been implemented in such a way that it operates independent of the fact base. Thus, no facts need to exist in the data base for the enhanced cycle checker to correctly verify the validity of potential cycles.

11.5.3 Explanation Facility

As part of the enhancement to the original cycle checker, an explanation facility which allows the developer to see why a cycle was reported as either genuine or phony was developed. The explanation facility is in the form of a graphical display and operates independently of the enhanced cycle checker. The developer selects a rule chain that was reported as either genuine or phony. For this rule chain, a pop-up window is then constructed.

If the cycle was reported as genuine, a possible chain of literals which validates the cycle is displayed in the pop-up window. In addition, it is also beneficial to report (for genuine cycles only) a range of legal values for each variable contained in the chain of literals, if such a range of values is defined. For example, if the RHS literal *p(X,Y)* of one rule in a genuine cycle unifies with the LHS literal *p(W,Z)* of the next rule in the cycle, we state a range of legal values for Y and Z for which the cycle still remains genuine, say, when Y = Z is bound by [0,10]. In addition, a set of possible instances and classes that X and W may assume is also given, say, when X = W is in [mark,gunner,pilot].

On the other hand, if the cycle is reported as phony, an explanation is written to the window notifying the developer of the source of the failure. The reason for a phony cycle may be the failure of a variable binding to propagate correctly or the failure of a check regarding a semantic constraint imposed on a literal. This facility allows the developer to browse one phony cycle at a time instead by being overwhelmed by information about all of the cycles at the same time.

CHAPTER 12

VALIDATION OF NONMONOTONIC REASONING

Ordinary logic is monotonic in nature. Any deduction from a base set of facts and conditions can only result in the addition of more facts to the fact-base. Nonmonotonic systems allow other revisions to an existing knowledge base state. As the result of adding or deleting a fact to the fact-base, other facts which may have been previously provable suddenly lose their support and consequently can no longer be considered true. Nonmonotonic reasoning appears in various guises in expert system shells. These include *delete* actions in RHS consequents of production rules, *negation by failure* in the LHS antecedents of rules, and *default values* for attributes in frame-based systems.

Many of the effects of nonmonotonic reasoning are already being handled by the checkers described in the previous sections; for instance, the conflict checker takes into account delete actions on the RHSs of rules. The checkers discussed in this section cover the effects of KEE's nonmonotonic *new.world.action (NWA) rules* which explicitly create new worlds based on previous ones.

12.1 ELIMINATION OF USELESS NEW.WORLD.ACTION RULES

In KEE, deduction rules which conclude *FALSE* can be used to eliminate inconsistent, improbable, or undesirable worlds. Worlds with *FALSE* in them will not be used any further by the rule system.

In order to prevent an action rule from generating a "bad" world, the conjunction of conditions in the new world action rule can not be more specific than the conditions of a *DEDUCE-FALSE* rule. This checker reports on those NWA rules that can not generate viable worlds, reusing some of the code that was originally written for the subsumption checker.

For example, consider the following two rules:

```
running_on_empty:

   ( IF
     ( NOT ( THE SPEED OF ?P IS 0))
     ( THE CURRENT_FUEL_AMOUNT OF ?P IS 0)
     THEN DEDUCE FALSE)

crashing_plane:

   ( IF
     ( ?P IS IN PLANE)
     ( THE CURRENT_FUEL_AMOUNT OF ?P IS 0)
     ( NOT ( THE SPEED OF ?P IS 0))
     THEN IN.NEW.WORLD
     ( DELETE
       ( ?P IS IN PLANE)))
```

The rule *running_on_empty* states that any time something is moving but has no fuel,

then a bad world results. The rule *crashing_plane* states that a plane that has no fuel and is moving should be deleted from the knowledge base. This checker notices that *running_on_empty* will fire every time that *crashing_plane* does. Because *running_on_empty* is a "DEDUCE-FALSE" rule, *crashing_plane* is rendered useless, so the knowledge engineer should modify or remove at least one of these rules.

12.2 DETECTION OF IMPOSSIBLE PLANS

In a "generate and test" problem, if a set of all test conditions is inconsistent, then no solution will be found, so it is essential that test conditions must be consistent. In KEE, this is equivalent to validating that the set of negations of *DEDUCE-FALSE* rules is consistent. This checker performs that computation. It negates a clause in disjunctive normal form, localizing the effects of negation to the literals. The resulting negated clause is amenable to checking by the residue-based backward chaining interference checker.

The control checkers as well as versions of the redundancy and rule-inconsistency checkers also use residue-analysis to uncover anomalies. In the residue analysis approach to verification, one generates the conditions necessary to infer a goal and then tries to determine whether those conditions are likely to occur simultaneously. A residue is simply a KB-state from which the rules can derive some goal; therefore, one must be careful not to require an inconsistent residue to demonstrate an anomaly. Inconsistent residues are useless because from an inconsistent set of beliefs, one in theory can prove anything. An example of an inconsistent residue is to assume within the residue that the literal *A* is absent/unprovable, whereas *A* can actually be proven from the other literals within the residue.

The backward chaining interference checker was originally designed to check for interference among the subgoals (literals) of a single rule. Because the negated clause to be tested is not found in any single rule, the backward chaining interference checker has been generalized to report knowledge bases where negated clauses exhibit strong interference — indicating that there are no possible viable new worlds via a declarative reading of the knowledge base.

Because this checker deals with negated clauses, DEVA must take extra care to ensure that the variables are bound correctly. In practice, most of the conditions of a "deduce false" rule are likely to be positive literals, (e.g., "member_of(Var, Class)") the negations of which (e.g., "cant_find(member_of(Var, Class))") will not create bindings during inference. This causes inaccuracy during inference.

Consider the example where the initial preconditions of the deduce_false rules are:

```
[member_of(X,'1_br_house'),has_wing(X,true)]
[member_of(Y,bird),not(has_wing(Y,true))]
```

which negates to:

```
Good_Conds = [or([cant_find(member_of(X,1_br_house)),cant_find(has_wing(X,true))]),
              or([cant_find(member_of(Y,bird)),has_wing(Y,true)])]
```

If we had a KB of:

```
member_of(my_hovel,'1_br_house'),  has_wing(my_hovel,false)
member_of(tweety,bird),            has_wing(tweety,true)
```

Good_Conds would be unsatisfiable, because of the semantics of cant_find.

To solve this problem, DEVA inserts a "member_of(Var, entities)" before each occurrence of "cant_find(member_of(Var, Class))" to provide the intended semantics for variable bindings.

For the example, the new procedure transforms Good_Conds into:

```
Goal_Conds = [member_of(X,entities),
              or([cant_find(member_of(X,1_br_house)),cant_find(has_wing(X,true))]),
              member_of(Y,entities),
              or([cant_find(member_of(Y,bird)),has_wing(Y,true)])]
```

which is satisfiable and has the intended semantics.

The user may choose whether to use facts during the formation of a residue. Using both facts and rules during residue creation more accurately indicates whether a given KB can give rise to unattainable goal conditions. Using rules alone indicates where trouble may arise due to a modification to the initial facts of a KB. For example, consider an initial KB consisting of:

¬a → deduce false
¬b → deduce false
¬a → b
a
b

The negation of the preconditions of the "deduce false" rules is "a ∧ b". Because we have both of the facts "a" and "b" in our KB, we can generate a "good" world. If we did not have the facts in our initial KB, we would have had to prove "a ∧ b" using the available rules. Because there is no rule concluding "a", we assume it. There is a rule which concludes "b", but to fire it requires us to assume "¬a", which contradicts our previous assumption.

This checker reports occasions of strong interference, i.e., when all possible proof trees exhibit interference. Because it would be impractical to report every possible conflicting proof, when the test conditions are inconsistent, a single representative inconsistent proof is shown. When the test conditions are consistent, a consistent proof is displayed

12.3 VALIDATION OF NEW.WORLD.ACTION RULES

Syntactically, new.world.action rules are similar to ordinary (same.world.action) rules. They have antecedents on the LHS and consequents on the RHS. As a result, they are prone to many of the errors of same.world.action rules — they may be unreachable, logically incomplete, etc.

Because new.world.action rules and same.world.action rules are syntactically similar, it was unnecessary to devise a multitude of checkers specialized to handle new.world.action rules — they may be validated using many of the checkers discussed in the previous sections. As an

example, consider the two rules mentioned in the section *Elimination of useless new.world.action rules* — *crashing_plane* and *running_on_empty*. Because the LHS of *crashing_plane* is more specific than that of *running_on_empty* and because they have different RHSs, these rules will be flagged by the ambiguity checker.

CHAPTER 13

CONCLUSION

13.1 DEVA, A GENERIC KBS VALIDATION SYSTEM

We have designed and implemented DEVA, a generic KB validation system which is independent of any particular ES shell or KBS application. This independence was obtained by providing a special translator for KEE which translates KBSs written in KEE into the DEVA metalanguage format upon which the DEVA validation modules operate. DEVA is thus theoretically applicable to the validation of KBSs written in other ES shells, such as ART (Automated Reasoning Tool, Inference Corporation), CLIPS (NASA's C-Language Integrated Production System), or others. We have demonstrated this genericness by validating CLIPS-based ESs provided by NASA Johnson Space Center with DEVA using our CLIPS translator developed under Lockheed's concurrent IR&D effort on Validation of Knowledge-Based Systems.

The DEVA approach is unique. No other automated KBS validation system is ES shell language, domain, and application independent. No other automated KBS validation system provides a generic and powerful metalanguage for expressing application requirements and constraints.

The DEVA metalanguage predicates are easy-to-use, powerful, and expressive; they allow the KBS developer/designer to specify requirements and constraints about objects, relations, rules, control, and behavior that his system must meet. All DEVA modules make use of DEVA's metalanguage statements.

13.2 THE SUPERIORITY OF DEVA'S LOGIC-BASED APPROACH

DEVA provides validation functionality which by far surpasses that of KEE or other first-generation ES shell languages. The V&V functionality provided by ES shell languages is typically limited to integrity checking, making sure that the values of instances of concepts, classes and binary relations, meet the requirements stated in the schema definition of the concept and its slots.

The DEVA approach to creating generic validation tools is based on the fact that the knowledge representation of rule-based and frame-based KBS shells is logic-based, though typically not full 1^{st}-order logic-based. Applications or models written in such shells are therefore amenable to formal validation using a full 1^{st}-order logic language (FOL) and mechanical theorem proving. The validation of such systems can be further improved by making use of a FOL enhanced with metaknowledge (higher-order knowledge) with semantic information and integrity constraints (EFOL). This permits the system's designer to formulate conditions, not expressible or efficiently enforceable in his ES shell language, that the rules, frames, facts, and other components of a domain model have to meet. The DEVA approach obtains this genericness through two components: a full EFOL-based metalanguage and ES shell language-specific translators. DEVA uses a metalanguage with many metapredicates to represent knowledge about knowledge. The metapredicates provide high-level generic data structures to represent not only all constructs of KEE, but also constructs of other current ES shells. Each validation requirement or constraint type is represented by a separate metavalidation predicate (with that name) which checks the KB for the adherence to the requirement/constraint. The DEVA metalanguage can therefore be easily extended to

provide additional, as yet not incorporated or discovered, validation criteria. The independence from a particular ES shell language is obtained by ES shell language-specific translators.

DEVA has the following significant advantages in comparison to ES shell languages and other validation systems. It is generic, independent of particular domains and applications, amenable for reuse of validation knowledge, easily extensible, and usable for the validation of shared or cooperating KBSs. DEVA is generic; it can validate KBSs written in any ES shell language once a translator for the ES shell language has been written. It is independent of particular domains and applications; knowledge required for the validation of domains and applications is simply added to DEVA's metaknowledge base. This theoretically also permits the reuse of validation knowledge for other applications in a particular domain based on a common representation of the validation vocabulary or by using the DEVA metalanguage predicates *synonymous* and *alias*, which establish substitutability for properties/relations and names. These primitives also permit the validation of shared KBs, once the equivalences of the different expressions and objects in the KBs to be shared has been declared to DEVA.

The development of DEVA demonstrates the superiority of a logic- and metaknowledge-based approach for the validation of KBSs written in ES shell languages.

Appendix A

DEVA USERS' MANUAL

Lockheed Missiles & Space Company, Inc.
Lockheed Software Technology Center, O/96-10, B/30E
2101 East St. Elmo Road Austin, Texas 78744

July 1990

This Manual was written for Rome Air Development Center under Contract F30602-88-C-0130.

A1. RESOURCE REQUIREMENTS

A1.1. Software

Unlike previous versions of DEVA, this version is a standalone executable, and does not need Quintus Prolog, or Quintus ProWINDOWS. The translator was built using Sun Microsystems Common Lisp version A2.1.3 and IntelliCorp Knowledge Engineering Environment (KEE) version 3.1.75. The operating system supported is SunOS 4.03.

A1.2. Hardware

Sun 4 or Sun 3
A3 button Sun mouse and pad
Color monitor

A2. INSTALLATION

A2.1. Installing DEVA

On the installation tape there is the installation script called:

```
installdeva
```

DEVA may be installed on either a Sun-3 or Sun-4 using installdeva. If you plan to use both of these machines you will need two different directories, and run installdeva separately from each of them.

1) Select a directory in your system hierarchy for the installation directory deva.2. We recommend placing it under /usr/local, although any directory will do. Unloading the tape in */usr/local* will create the directory:

   /usr/local/deva.2

   If you wish to use a directory other than /usr/local, substitute that directory for all occurrences of /usr/local in the following instructions.

2) The tape you received was written using tar. When unloading the tape, you will need write permission on the directory you choose and read permission for the tape device. This may be best achieved by being logged in as *root* when you unload the tape. To unload DEVA, load the tape into your tape drive and type:

   ```
   cd /usr/local
   tar xvf /dev/rmt0 deva.2
   ```

   where /dev/rmt0 should be the SunOS device name for the tape drive on your system. This name shown here follows the normal SunOS device naming convention. If your system is different, use your particular device name in place of /dev/rmt0. The tar command will list all the files it unloads.

3) Usually, all files in the release will be owned by *bin*. To correctly access these files during installation, and to make sure any generated files are owned also by *bin*, you should

change your user-id to *bin* once the tape has been unloaded. The following command can be used, provided that your are already logged in as *root:*

su bin

4) Having unloaded DEVA from tape, you are now ready to do the actual installation. The installation procedure described below produces a subdirectory of */usr/local/deva.2* with the executable files which are tailored to your precise hardware and UNIX operating system 4.03.

   To install DEVA, first go to the installation directory:

   cd /usr/local/deva.2

   and then type:

   installdeva

5) To make DEVA generally available, the file deva from usr/local/deva.2 should be *copied or linked* to a standard bin area on your system. Only that file needs to be copied; the other files should remain where they are.

   To use copy, type:

   cp /usr/local/deva.2/deva /usr/local/bin

   or to use link:

   ln -s /usr/local/deva.2/deva /usr/local/bin

   or

   ln /usr/local/deva.2/deva /usr/local/bin

A2.2. Installing the KEE Translator

There are three main files that system support will have to modify to install the KEE Translator:

1) *kee-startup.lisp* — a KEE system file under /usr/local/kee/misc/lisp where KEE resides on the system

2) *A-bin.lisp* — a DEVA file that comes with the KEE Translator

3) *uncompiled-VARS.lisp* — a DEVA file that comes with the KEE Translator.

A2.2.1 Modifying the KEE system files

1) At the bottom of the file, kee-startup.lisp, type in the following LISP code, inserting the directory path where you have the file A-bin.lisp stored:

```
(if (probe-file "/< your path> /A-bin.lisp")
    (load "/< same path as above> /A-bin.lisp"))
```

2) In A-bin.lisp change all paths to the pathname where you will store the KEE Translator. In uncompiled-VARS.lisp make the following modifications:

set %%&deva-translator-path to "/< your path> /A-bin.lisp"

A3. USERS' GUIDE

A3.1 Running DEVA and KEE from the same Desktop

This section assumes some knowledge of the Sun Windowing System known as Sunview or Suntools. If unfamiliar with Sunview or Suntools, read the Sunview Manual before proceeding. From the UNIX prompt type:

```
sunview
```
or
```
suntools
```

To load DEVA, open a shell window and enter:

```
deva.exe
```

To load KEE, from another shell window type

```
kee
```

When the KEE desktop appears, it will take several minutes to load the ADVICE system (used by the DEVA translator). During the KEE loading process, the user will be prompted in the lower left corner typescript window:

Do you wish to load the Translator (y/n)?

The user should enter *Y*, and hit return.

A4. USER INTERFACE

Consult Chapter 3 sections 3.1-3.2

A5. GRAPHICS

A5.1 Connection Graph

Consult Chapter 3 section 3.3.1 Connection Graph

A5.2 Rule browser

Consult Chapter 3 section 3.3.2 Rule Browser

A5.3 Unit Graph

Consult Chapter 3 section 3.3.3 Unit Graph

A6. TUTORIAL

This is a step by step tutorial on how to bring up DEVA, load a test knowledge base (KB) from

1) the KEE environment

2) a saved KEE translation file

3) a KB saved in DEVA format

A6.1 The following KBs are supplied

1) flight A simple KB about military aircraft

2) circuit A full adder

3) myequals The transitivity of equals is explored

4) triage Some rules covering the Emergency Room

All KBs in KEE format, KEE messages format, and DEVA format are stored in the subdirectory *cases*.

A6.2 Loading a KB into DEVA

A6.2.1 Loading a KB from KEE

First start the KEE and DEVA processes as described in Tutorial section above. In KEE, move the mouse to the KEE key icon in the upper left hand side of the display. Left-click to bring up a KEE KEY menu (the third icon in the upper left hand corner of the KEE window). Select load KB option. Bring the cursor down to the Typescript Window where you are being prompted for the name fo the KB you want to load. Type:

cases/flight.u

When the KB has finished loading in KEE, a message will be printed in the KEE Typescript Window

Knowledge base FLIGHT in /< your-path> /cases/flight.u loaded.

To load the KB from KEE to DEVA, left-click on

FLIGHT

in the upper left window Knowledge Bases. A KEE popup menu will appear called KB Commands. Select

Send KB to DEVA

When a KB is saved, a translation file of KEE messages will be constructed and saved with the extension < KB name> -kee.msg as in *flight-kee.msg.*

To load a KB in KEE into DEVA, first move the cursor to the DEVA window, and left-click on the data circular button in the options window so that is says *KEE Messages.* Then in the KEE window left-click on the KB name in the *Knowledge Bases* window. A submenu will appear called *KB Commands.* Left-click on the menu option *Send KB to DEVA.* In the DEVA Report Window, a message will appear that lets the user know that the translation/loading process has begun.

A6.2.2 Loading a KB saved in DEVA format

First move the cursor to the DEVA window and left-click on the data circular button in the options window so that is reads *DEVA Format.* Then in the DEVA window, left-click on the load button. A browser will appear on the right hand side of the screen listing the KBs in the cases directory with a .pl extension, as in *flight.pl.* Move the mouse the browser, and select the name *flight.pl.* In the DEVA Report Window, you should see a message reporting the KB being loaded.

A6.2.3 Loading a KB saved in KEE Message format

First move the cursor to the DEVA window and left-click on the data circular button in the options window so that is reads *KEE Messages.* Then in the DEVA window, left-click on the load button. A browser will appear on the right hand side of the screen listing the KBs in the cases directory with a *-kee.msg extension,* as in *flight-kee.msg* Move the mouse to the browser, and select *flight-kee.msg* In the DEVA Report Window, you should see a message reporting the KB being loaded.

APPENDIX B

VALIDATION OF FRAME-BASED SYSTEMS

The value of the EVA validation methodology lies in the fact that the validation modules developed according to it are applicable not only to rule-based systems but also to frame-based systems. The reason for this is that logic-based and frame-based systems are equivalent. Either can be represented in the formalism of the other. Patrick Hayes (1979) already showed this for frame-based systems without 2^{nd}-order primitives, such as *is-a-subset-of*, for example. Most current frame-based systems are variants of semantic nets whose representation in a logic formalism requires 1^{st}-order logic with a few 2^{nd}-order primitives.

Since we have heard repeatedly — even from cognoscenti in validation of knowledge-based systems — that EVA (the Lockheed IR&D precursor of DEVA) applies only to rule-based, i.e., logic-based systems, and not to frame-based systems, we shall simply show how KEE, an admittedly frame-based system, can be translated into the EVA representation, a representation which uses 1^{st}-order and 2^{nd}-order logic.

A KEE KB is a collection of *units* (frames), which are organized in a hierarchical class structure. A unit may denote an object, a class of objects, a relation, a relational fact, a rule, or a class of rules. Each unit has a set of *slots* whose *values* represent the unit's current state. In addition to a value, a slot can also have attached *facets* (restrictions on values).

To repeat: The basic construct in KEE is a unit. A unit may represent or model a level-0 or a level-1 entity or term. For example, in

father-of(g000009,g000082)
person(g000009)
adult(g000082)

g000009 and *g000082* denote (model) entities in the real world, let us say Harry S. Truman and Margaret Truman. *g000009* and *g000082* are thus level-0 terms. Rather than using meaningless surrogates like *g000009* and *g000082*, expert system shells allow meaningful level-0 representations like

father-of(Harry_S_Truman,Margaret_Truman)
person(Harry_S_Truman)
adult(Margaret_Truman)

In an expression of the form *a(b)* or *a(b,...)*, the term *a* is a level-n term where *n* is higher by one than the highest level of any term in *(b)* or *(b,...)*. In our examples *father-of*, *adult*, and *person* are therefore level-1 terms.

In expressions like *slot-of(person,age)*, *class(person)*, *subset-of(person,adult)*, the terms *slot-of*, *class*, *subset-of* are level-2 terms since *person*, *age*, *adult* are level-1 terms.

Note that some level-1 expressions can also be represented by means of level-2 expressions. Thus some shells use *person(Harry_S_Truman)* whereas others prefer *member-of(person,Harry_S_Truman)* or *instance-of(person,Harry_S_Truman)* to express that Harry S. Truman is a person.

In general, a formalism using terms at level-n can be represented in an n^{th}-order logic.

KEE-Units

KEE supports units at three levels. Level-0 units are objects which **are** members (instances) of a class, but are not themselves a class, i.e., have no members themselves; they are represented as individuals or constants in 1^{st}-order predicate logic. Level-2 units are collections which **have** members, and which may have superclasses and/or subclasses. (They are also themselves members [instances] of the KEE unit *classes.*) They are represented as predicates in 1^{st}-order logic or 2^{nd}-order constants in a 2^{nd}-order logic. Level-3 units are KEE-system-specific objects, denoted by KEE's reserved primitives, such as *classes*, *entities*, *slot*, *generic units*, *superclasses*, *member of*, and others; they are represented as predicates in a 2^{nd}-order logic. KEE also treats rules as units. In the context of this discussion, we take their translatability for granted.

The translation of KEE terms into formulas, i.e., facts, of 1^{st}-order and 2^{nd}-order logic, is straightforward. (In the following translations, we will represent level-0 terms in all lower-case, level-1 terms with initial capitalization, level-2 terms in all upper-case.)

A level-0 unit becomes a 1^{st}-order constant; its predicate is the KEE term appearing after MEMBER OF.

KEE	EVA
UNIT: cutty.sark MEMBER OF: Square.rigged.ships	Square.rigged.ships(cutty.sark)

A level-1 unit becomes a 2^{nd}-order constant; its predicate is the KEE term CLASSES.

KEE	EVA
UNIT: Square.rigged.ships MEMBER OF: CLASSES	CLASSES(Square.rigged.ships)

KEE SUPERCLASS information can be represented by a rule in logic. It is represented by ISA in EVA.

KEE	EVA
UNIT: Square.rigged.ships MEMBER OF: CLASSES SUPERCLASSES: Commercial.ships, Sailing.ships	(x) Square.rigged.ships(x)→Commercial.ships(x) (x) Square.rigged.ships(x)→Sailing.ships(x) ISA(Square.rigged.ships,Commercial.ships) ISA(Square.rigged.ships,Sailing.ships)

Slots

KEE distinguishes two kinds of slots: *own-slots* and *member-slots*. A level-0 KEE unit can only have own-slots; a level-1 KEE unit may have both own-slots and member-slots. Own-slots cannot be inherited. Own-slots of a level-0 unit are level-1 predicates;

own-slots of level-1 units are level-2 predicates. Member-slots are also level-1 predicates. They are inheritable. When they are inherited by a level-0 unit, they change into own-slots. In other words, a level-0 unit has only own-slots and no member-slots.

KEE own-slots of level-0 units become 1st-order predicates, own-slots of level-1 units become 2nd-order predicates; their first argument is the KEE unit, their second, the value(s) of the KEE term *VALUES*.

KEE	EVA
UNIT: cutty.sark MEMBER OF: Square.rigged.ships OWN SLOT: Cargo.carried FROM Ships VALUES: steel	Cargo.carried(cutty.sark,steel)
UNIT: Square.rigged.ships MEMBER OF: CLASSES SUPERCLASSES: Commercial.ships, Sailing.ships OWN SLOT: MOST.VALUABLE.CARGO VALUES: gold	MOST.VALUABLE.CARGO(Square.rigged.ships)

KEE member-slots result in 1st-order **rules**.

KEE	EVA
UNIT: Square.rigged.ships MEMBER OF: CLASSES SUPERCLASSES: Commercial.ships, Sailing.ships MEMBER SLOT: Crew FROM Sailing.ships VALUES: unknown	(x)Ey (Square.rigged.ships(x)→Crew(x,y))

Facets

KEE slots have *facets* which mostly provide information about requirements and constraints that slot values need to satisfy. Of importance for translation into logic are the facets *Valueclass*, *Cardinality.max*, and *Cardinality.min*. The KEE facet *inheritance* contains information for KEE on how to propagate member-slot data from a class to its subclasses and members. *Inheritance* requires no special translation mechanism since the inherited information is available at each such subclass and member; i.e., it will be translated as a *member-slot* (for classes and subclasses) or *own-slot* (for members). The KEE facet *comments* is for recording user comments and is not translated into logic. All KEE facets of interest are translated into 2nd-order facts in a straightforward manner.

The specification operator of the KEE facet *ValueClass* is translated into the corresponding EVA level-2 predicate whose first argument is the KEE slot name, followed by the name of the KEE unit, followed by the value(s) of the KEE ValueClass.

KEE	EVA
UNIT: Bird.type	
MEMBER OF: CLASSES	
MEMBERSLOT: Selected.bird FROM Bird.type	
VALUECLASS: (NOT.ONE.OF green.finch kestrel black.backed bald.eagle)	ILLEGAL_VALS(Selected.bird,Bird.type,[green.finch, kestrel, black.backed, bald.eagle])

The KEE facets *cardinality.min* and *cardinality.max* are combined into the EVA level-2 predicate NUMERICAL_RANGE.

KEE	EVA
UNIT: Birds	
MEMBER OF: LOCAL.FAUNA, CLASSES	
MEMBERSLOT: Local.types FROM Birds	
VALUECLASS: (LIST)	
CARDINALITY.MIN: 3	NUMERICAL_RANGE(Local.types, Birds, 3, 7)
CARDINALITY.MAX: 7	

APPENDIX C

A BRIEF HISTORY OF VALIDATION OF KNOWLEDGE-BASED SYSTEMS

It is evident, particularly in the Department of Defense, that KBSs cannot gain wide acceptance without some formal method of proving the correctness of an application KB. Since KBSs are usually developed without any pre-written requirements, it is not clear what validations should be performed for them. In the early attempts of validating KBSs, people came up with the general concepts of consistency and completeness. These concepts may not be explicit requirements of any particular KBS, rather, it is a general belief that if a KBS is not consistent or complete, it probably will not work correctly, at least for certain cases. These concepts are useful even though there are no standard definitions of them. For example, some authors restrict their definitions to rule pairs, others to propositional rules, i.e., rules without variables.

Suwa et al. [1982] deal with *consistency* and *completeness* of rules written in ONCOCIN, an EMYCIN-like expert system for oncology. To speed up the checking process, related rules are clustered and checked together.

Nguyen et al. [1987] deal with *inconsistency* involving pairs of rules, and consider *dead-end rules* and *unreachable literals* as some kind of *incompleteness*. They also detect equivalent predicates by detecting *cycles* in two rules.

Cragun and Steudel [1987] use a method based on logical decision table checking to determine the *completeness* and *consistency* of KBs. They also limit themselves to propositional logic, i.e., rules without variables. Their method constructs a decision table for all rules in the KB. Each row of the table is either a condition or an action, and each column denotes a rule. There is an entry at row *i* and column *j* if rule *j* has the condition or action at row *i*. *Ambiguity* occurs when the same set of logical conditions satisfy two or more different rules that have different actions. *Redundancy* occurs when the same logical conditions satisfy more than one rule, but the actions are the same. *Completeness* is present when all possible combinations of logic are addressed by the rules in the table.

Ginsberg [1988] generates sum-of-products (disjunctive normal form) for each conclusion to check KBs for *inconsistency* and *redundancy*. Each product represents possible input data. A product is redundant if it is subsumed by other products. A product is inconsistent if it contains conflicting facts. This approach is restricted to rules of propositional logic, i.e., without variables.

The most common validation approach deals with the functional behavior of KBSs. Since it is often very hard to specify the functional requirements of a system, the KBS is used as an approximation of the functional specification of the system. In this approach, the KBS is tested (evaluated) by running the KBS on a given collection of cases of input data. The KBS-generated output data are compared with the given (expected) output data. If there are any discrepancies, the rules in the KBS are refined. If the expected output data are not readily available, the KBS generated output data are evaluated by inspection. Davis [1979], Politakis and Weiss [1980, 1984], Weiss and Kulikowski [1983], and Ginsberg and Weiss [1985] all take this approach. (The Rule Refiner of DEVA will give the developer similar capability.)

At the Lockheed Artificial Intelligence Center, we started an Independent Research project on Validation of Knowledge-based Systems in 1986. Our goal was to develop flexible KBS

validation tools. We achieve this goal by using the **meta-knowledge approach** [Stachowitz et al. 1987a, 1987b, 1987c]. Meta-knowledge, or knowledge about knowledge, describes constraints on the knowledge that can be used for *redundancy*, *consistency*, *completeness* and *correctness* checking. The power of using metaknowledge derives from the fact that validation criteria for applications cannot be standardized: each application has its idiosyncratic statements that need to be validated, but these validation criteria can be formulated and represented by means of the meta-predicates.

DISTRIBUTION LIST

addresses	number of copies
RL/C3CA ATTN: Robert N. Ruberti Griffiss AFB NY 13441-5700	10
Lockheed Missiles and Space Company Software Technology Center 2100 E. St. Elmo Rd Austin TX 78670	5
RL/DOVL Technical Library Griffiss AFB NY 13441-5700	1
Administrator Defense Technical Info Center DTIC-FDAC Cameron Station Building 5 Alexandria VA 22304-6145	2
Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington VA 22209-2308	2
RL/C3AB Griffiss AFB NY 13441-5700	1
HQ USAF/SCTT Washington DC 20330-5190	1
SAF/AQSC Pentagon Rm 4D 269 Wash DC 20330	1

Naval Warfare Assessment Center 1
GIDEP Operations Center/Code 30G
ATTN: E Pichards
Corona CA 91720

HQ AFSC/XTH 1
Andrews AFB MD 20334-5000

HQ SAC/SCPT 2
OFFUTT AFB NE 68046

HQ TAC/DRIY 1
ATTN: Maj. Divine
Langley AFB VA 23665-5575

HQ TAC/DOA 1
Langley AFB VA 23665-5554

ASD/ENEMS 1
Wright-Patterson AFB OH 45433-6503

WRDC/AAAI-4 1
Wright-Patterson AFB OH 45433-6543

WRDC/AAAI-2 1
ATTN: Mr Franklin Hutson
WPAFB OH 45433-6543

AFIT/LDEE 1
Building 642, Area B
Wright-Patterson AFB OH 45433-6583

WRDC/MTEL 1
Wright-Patterson AFB OH 45433

AAMRL/HE 1
Wright-Patterson AFB OH 45433-6573

Air Force Human Resources Lab 1
Technical Documents Center
AFHRL/LRS-TDC
Wright-Patterson AFB OH 45433

AUL/LSE 1
Bldg 1405
Maxwell AFB AL 36112-5564

HQ ATC/TTOI 1
ATTN: Lt Col Killian
Randolph AFB TX 78150-5001

AFLMC/LGY 1
ATTN: Maj. Shaffer
Building 205
Gunter AFS AL 36114-6693

US Army Strategic Def 1
CSSD-IM-PA
PO Box 1500
Huntsville AL 35807-3801

Commanding Officer 1
Naval Avionics Center
Library D/765
Indianapolis IN 46219-2189

Commanding Officer 1
Naval Ocean Systems Center
Technical Library
Code 96428
San Diego CA 92152-5000

Cmdr 1
Naval Weapons Center
Technical Library/C3431
China Lake CA 93555-6001

Superintendent 1
Code 524
Naval Postgraduate School
Monterey CA 93943-5000

Space & Naval Warfare Systems Comm 1
Washington DC 20363-5100

CDR, U.S. Army Missile Command 2
Redstone Scientific Info Center
AMSMI-RD-CS-R/ILL Documents
Redstone Arsenal AL 35898-5241

Advisory Group on Electron Devices 2
Attn: Documents
2011 Crystal Drive, Suite 307
Arlington VA 22202

Los Alamos National Laboratory 1
Report Library
MS 5000
Los Alamos NM 87544

AEDC Library
Tech Files/MS-100
Arnold AFB TN 37389 1

Commander, USAG
ASQH-PCA-CRL/Tech Lib
Bldg 61801
Ft Huachuca AZ 85613-6000 1

1839 EIG/EIT
Keesler AFB MS 39534-6348 1

AFEWC/ESRI
San Antonio TX 78243-5001 3

SEI JPO
ATTN: Major Charles J. Ryan
Carnegie Mellon University
Pittsburgh PA 15213-3890 1

Director NSA/CSS
W157
9800 Savage Road
Fort Meade MD 21055-6000 1

NSA
ATTN: D. Alley
Div X911
9800 Savage Road
Ft Meade MD 20755-6000

1

Director
NSA/CSS R12
ATTN: Mr. Dennis Heinbuch
9800 Savage Road
Fort George G. Meade MD 20755-6000

1

DoD
R31
9800 Savage Road
Ft. Meade MD 20755-6000

1

DIRNSA
R509
9800 Savage Road
Ft Meade MD 20775

1

Director
NSA/CSS
R08/P & E BLDG
Fort George G. Meade MD 20755-6000

1

DOD Computer Center
C/TIC
9800 Savage Road
Fort George G. Meade MD 20755-6000

1

ESD/AV
HANSCOM AFB MA 01731-5000

1

ESD/IC
HANSCOM AFB MA 01731-5000

1

FL 2807/RESEARCH LIBRARY 1
OL AA/SULL
HANSCOM AFB MA 01731-5000

Technical Reports Center 1
Mail Drop D130
Burlington Road
Bedford MA 01731

Software Engr'g Inst Tech Library 1
ATTN: Mr Dennis Smith
Carnegie Mellon University
Pittsburgh PA 15213-3890

Software Options, Inc. 1
ATTN: Mr Tom Cheatham
22 Hilliard Street
Cambridge MA 02138

USC-ISI 1
ATTN: Dr Robert M. Balzer
4676 Admiralty Way
Marina del Rey CA 90292-6695

Kestrel Institute 1
ATTN: Dr Cordell Green
1801 Page Mill Road
Palo Alto CA 94304

Rochester Institute of Technology 1
ATTN: Prof J. A. Lasky
1 Lomb Memorial Drive
P.O. Box 9887
Rochester NY 14613-5700

Westinghouse Electronics Corp 1
ATTN: Mr Dennis Bielak
Electronics Systems Group
P.O. Box 746, Mail Stop 432
Baltimore MD 21203

AFIT/ENG 1
ATTN: Paul Bailor, Major, USAF
WPAFB OH 45433-6583

Incremental Systems Corporation 1
ATTN: Dr Deborah A. Baker
319 South Craig Street
Pittsburgh PA 15213

The MITRE Corporation 1
ATTN: Mr Edward H. Bensley
Burlington Rd/Mail Stop A350
Bedford MA 01730

Univ of Illinois, Urbana-Champaign 1
ATTN: Sanjay Bhansali
Dept of Computer Sciences
1304 West Springfield
Urbana IL 61801

The MITRE Corporation 1
ATTN: Ms Melissa P. Chase
Burlington Road
Bedford MA 01730

Andersen Consulting 1
ATTN: Dr Michael E. DeBellis
100 South Wacker Drive
Chicago IL 60606

Univ of Illinois, Urbana-Champaign 1
ATTN: Dr Mehdi Harandi
Dept of Computer Sciences
1304 W. Springfield/240 Digital Lab
Urbana IL 61801

Honeywell, Inc. 1
ATTN: Mr Bert Harris
Federal Systems
7900 westpark Drive
McLean VA 22102

Lockheed Sanders, Inc. 1
ATTN: Mr David R. Harris
Information Systems/NCAI1-2232
95 Canal Street
Nashua NH 03061

Software Engineering Institute 1
ATTN: Mr William E. Hefley
Carnegie-Mellon University
SEI 2218
Pittsburgh PA 15213-38990

University of Southern California 1
ATTN: Dr W. Lewis Johnson
Information Sciences Institute
4676 Admiralty Way/Suite 1001
Marina del Rey CA 90292-6695

Columbia Univ/Dept Computer Science 1
ATTN: Dr Gail E. Kaiser
450 Computer Science Bldg
500 West 120th Street
New York NY 10027

Software Engineering Institute 1
ATTN: Kyo Chul Kang
Carnegie-Mellon University
Pittsburgh PA 15213-3890

Odyssey Research Associates, Inc. 1
ATTN: Dr Tanya Korelsky
301A Harris B. Dates Drive
Ithaca NY 14850-1313

Software Productivity Consortium 1
ATTN: Mr Robert Lai
2214 Rock Hill Road
Herndon VA 22070

AFIT/ENG 1
ATTN: Dr Gary B. Lamont
School of Engineering
Dept Electrical & Computer Engrg
WPAFB OH 45433-6583

McDonnell Douglas Space Station Co 1
ATTN: Ms Penny Muncaster-Jewell
Engineering Services
16055 Space Center Boulevard
Houston TX 77062-6208

NSA/Ofc of Research 1
ATTN: Ms Mary Anne Overman
9800 Savage Road
Ft George G. Meade MD 20755-6000

The MITRE Corporation 1
ATTN: Mr Howard Reubenstein
Burlington Road
Bedford MA 01730

AFSC/SCRX 1
ATTN: Ms Suzanne Rhoad
Andrews AFB MD 20334-5000

Andersen Consulting 1
ATTN: Dr William C. Sasso
Center for Strategic Tech Rsch
100 South Wacker Drive
Chicago IL 60606

AT&T Bell Laboratories 1
ATTN: Mr Peter G. Selfridge
Room 3C-441
600 Mountain Ave
Murray Hill NJ 07974

Vitro Corporation 1
ATTN: Mr Robert A. Small
14000 Georgia Avenue
Silver Spring MD 20906-2972

Odyssey Research Associates, Inc. 1
ATTN: Ms Maureen Stillman
301A Harris B. Dates Drive
Ithaca NY 14850-1313

WRDC/AAAF-3 1
ATTN: James P. Weber, Capt, USAF
Aeronautical Systems Division
WPAFB OH 45433-6543

Texas Instruments Incorporated 1
ATTN: Dr David L. Wells
P.O. Box 655474, MS 238
Dallas TX 75265

Boeing Computer Services 1
ATTN: Dr Phil Newcomb
MS 7L-64
P.O. Box 24346
Seattle WA 98124-0346

Lockheed Software Tehnology Center 1
ATTN: Mr Henson Graves
Org. 96-10 Bldg 254E
3251 Hanover Street
Palo Alto CA 94304-1191

McDonnell Douglas Corporation 1
ATTN: Dr Douglas Abbott
Dept 052 MC 306 9042
P.O. Box 516
St Louis MO 63042-0516

Reasoning Systems 1
ATTN: Dr Gordon Kotik
3260 Hillview Avenue
Palo Alto CA 94304

Rockwell International 1
ATTN: Ms Anne Stanley
3370 Miraloma Avenue
Anaheim CA 92803-3105

Texas A & M University 1
ATTN: Dr Paula Mayer
Knowledge Based Systems Laboratory
Dept of Industrial Engineering
College Station TX 77843

Honeywell Systems and Rsch Center 1
ATTN: Mr Aaron Larson
3660 Technology Drive
Minneapolis MN 55418

Kestrel Development Corporation 1
ATTN: Dr Richard Jullig
3260 Hillview Avenue
Palo Alto CA 94304

Aerospace Corporation 1
ATTN: Dr. Kirstie Bellman
M1/102 Computer Sci & Tech Subdiv
P. O. Box 92957
Los Angeles CA 90009-2957

Lockheed O/96-10 B/254E 1
ATTN: Jacky Combs
3251 Hanover Street
Palo Alto CA 94304-1191

DARPA/ISTO 1
ATTN: Steve Cross
1400 Wilson Blvd
Arlington VA 22209

NASA/Johnson Space Center 1
ATTN: Chris Culbert
Mail Code PT4
Houston TX 77058

Advanced Decision Systems 1
ATTN: Dr. Ted Linden
1500 Plymouth Street
Mountain View CA 94943-1230

SAIC 1
ATTN: Lance Miller
MS T1-6-3
PO Box 1303 (or 1710 Goodridge Dr)
McLean VA 22102

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.